Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Métricas para determinar la dificultad de sabotear una Rede Neuronal Convolucional

Estudante: María Taboada Pena
Dirección: Brais Cancela Barizo,
Verónica Bolón Canedo

A Coruña, febreiro de 2021.

A mi madre.

Agradecimientos

A mi abuela y a mi madre que querían verme con un título universitario. A mis tutores sin los que este trabajo no habría sido posible. A María y a Nuria pendientes en todo momento del avance de este trabajo. A Santi, sin el, probablemente no estaría en este grado, por estar durante el proceso y por darme la confianza que no tenía. Y a Miguel, que cree en mi, un paso más cerca.

Resumen

En este trabajo fin de grado es un proyecto de *Deep Learning*, en concreto trata del estudio de una de las vulnerabilidades de hacer clasificación con redes neuronales convolucionales (CNN), los ataques adversos.

Los ataques adversos cambian la imagen de entrada ligeramente de forma imperceptible para el ojo humano. Al pasar esta imagen modificada por el clasificador, la clasificación es errónea y con un porcentaje alto de confianza en la clase predicha.

El problema está más cercano a un problema de "hacking". Se pretende conocer la cantidad de modificaciones que hay que realizar en una imagen de entrada para que el clasificador cambie su predicción. A lo largo de la memoria se presenta un nuevo método, DENA, para mejorar la cuantificación de la robustez de las CNN.

Con este trabajo se pretende dar a conocer estos ataques, reproducirlos y dar un valor numérico a estos cambios con el fin de poder medir la robustez de una red ante posibles ataques de este tipo.

Abstract

This final degree project is about Deep Learning, specifically about the study of one of the vulnerabilities of classifying with convolutional neural networks (CNN): the adversarial attacks.

Adversarial attacks change the input image slightly imperceptibly to the human eye. When passing this modified image through the classifier, the classification is wrong and with a high percentage of confidence in the predicted class.

The problem is closer to a "hacking" problem. It is try to know the amount of modifications that must be made in an input image for the classifier to change its prediction. Throughout the memory, a new method, DENA, is presented to improve the quantification of the robustness of CNNs.

This work is intended to make these attacks known, reproduce them and give a numerical value to these changes in order to be able to measure the robustness of a network against possible attacks of this type.

Palabras clave:

- Ejemplo adverso
- Gradiente
- Función de pérdida
- Optimizador
- Red neuronal profunda

- Red neuronal convolucional

Keywords:

- *Adversal example*
- *Gradient*
- *Loss funtion*
- *Optimizer*
- *Deep Neural Network*
- *Convolutional Neural Network*

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introdución

Con los últimos avances en algoritmia, potencia computacional y *big data*, la inteligencia artificial (y el aprendizaje automático en particular) ha conseguido hitos muy importantes en los últimos años. Clave para estos avances han sido el uso de modelos de aprendizaje profundo, que están basados en redes de neuronas. Concretamente, a partir del 2012, una familia de modelos de aprendizaje automático llamada *ConvNets* [1] comienza a tener rendimiento en tareas de reconocimiento visual.

1.1 Viabilidad e impacto

De forma imperceptible e inconsciente, las redes neuronales están en un sinfín de tareas que realizamos a diario. Algunos ejemplos superficiales pueden ser la recomendación de películas en *Netflix* o la identificación y categorización de correos electrónicos. Pero también triunfan en campos que puede tener más repercusión si hay un *hackeo* como puede ser una red que diagnostique la COVID-19, rastrear vendedores de drogas en redes sociales, o incluso las *fake news*.

Un caso sencillo de *adversarial attack* podemos encontrarlo en los comienzos de la detección de *spam*. Los clasificadores obtuvieron mucho éxito contra correos que contenían textos como *¡Haga dinero rápido!, Refinancia tu hipoteca,...* . En un primer momento los atacantes convirtieron su texto claro por un texto lleno de signos de puntuación como *¡H4G4 D|nero r4p1do!,...*, cambiaban código *HTML*, cambios en los *tags*, convirtieron el texto a imágenes como en la Figura 1.1, de forma que incluso un sistema basado en análisis de texto no lo detectase. Como defensas, se tomaron medidas basadas en *hash* de imágenes previamente conocidas como *spam* utilizando *OCRs* para extraer los textos de las imágenes. Para evitar estas defensas, los atacantes comenzaron a su vez a aplicar transformaciones de imágenes con ruido aleatorio dificultando la tarea de reconocimiento de caracteres en la imagen, es decir, un ataque adverso.

Figura 1.1: Ejemplo de ataque en la clasificación de *spam*

En el caso de Netflix, este usa un sistema de clasificación con técnicas de aprendizaje máquina para recomendar películas basado en usuarios que han visto la misma película o serie y no solo hace *matching* con el título de la película, sino que hace *clipping* del *frame* que más se ajusta al perfil de usuario según los géneros que ha consumido (ver Figura 1.2).

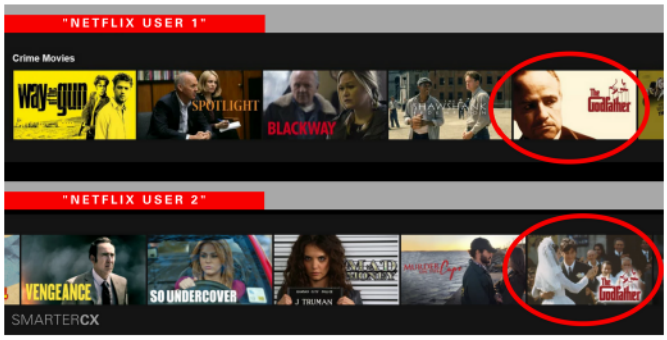


Figura 1.2: Ejemplo de recomendación de *Netflix*

Probablemente el usuario 1 tiene un perfil de películas de acción mientras el usuario 2 tiene un perfil más acorde al género romántico. Pero, ¿que pasaría si ese *frame* fuera modificado por un ataque adverso? Pongamos un ejemplo:

Tenemos un perfil de *Netfix* que está dedicado a un género infantil para no ver en él recomendaciones de películas para adultos. A un niño de 5 años le ponemos la película de "Bichos" y le dejamos viendo la película. Mientras tanto, un ataque adverso modifica un *frame* elegido para recomendar la siguiente película a ver. Un parámetro de la recomendación será ese *frame*, pero también habrá otros como por ejemplo las palabras claves en el título. Si ambos parámetros tienen una alta importancia en la ponderación de la recomendación final, quizás a nuestro niño, después de ver la película de "Bichos" le recomiende ver "El ciempiés humano" (ver Figura 1.3).

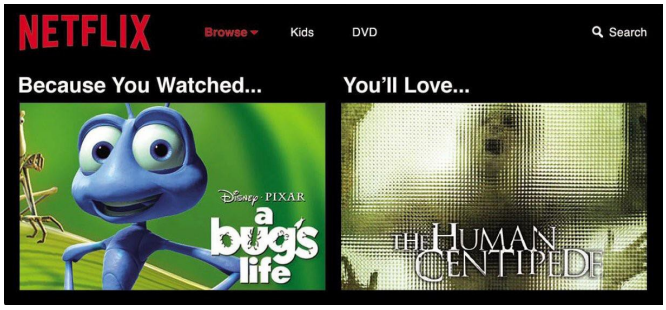


Figura 1.3: Recomendación adversaria de *Netflix*

Un caso de ejemplo adverso en el ámbito de la medicina puede ser el detectar si un lunar es benigno o maligno basándonos en fotografías del lunares (ver Figura 1.4). Después de aplicar a la imagen original el ruido del ataque, la red neuronal pasa de predecir un lunar benigno a uno maligno, cuando a simple vista, es imposible detectar el cambio.

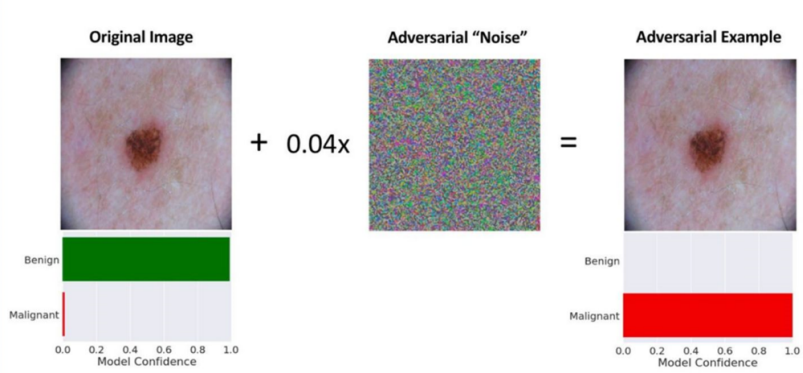


Figura 1.4: *Adversarial attack* en el diagnóstico de lunares

Si pensamos en ataques que puedan afectar a largo plazo, se podría pensar en la conducción autónoma: investigadores de *Keen Labs* han logrado generar imágenes adversarias alterando el sistema de piloto automático de los coches de Tesla [2]. En ataques de tipo *White-box* (ver sección 2.2.1) se podrían generar *adversarial examples* con una tasa de éxito de promedio del 98% [3]. Esto implica una alta susceptibilidad en proyectos de *self-driving open-source* como *comma.ai* [4], donde se exponen por completo la arquitectura y parámetros de los modelos [5]. *Waymo*, una empresa desarrolladora de vehículos autónomos pertenecientes al conglomerado *Alpahbet Inc* expone una serie de datos sacados de las imágenes capturadas por los sensores de alta resolución de sus automóviles en una amplia variedad de condiciones, con el fin de ayudar a la comunidad investigadora a avanzar en esta tecnología [6]. Estos datos podrían ser utilizados para entrenar una amplia variedad de modelos y generar *adversarial attacks* que en algunos casos podrían tener efecto en las redes utilizadas por *Waymo*.

También hay que tener en cuenta que hay una gran distancia entre engañar a un modelo y engañar a un sistema que contiene un modelo. De forma que muchas veces las redes neuronales solo son un sistema más que interactúa con los distintos componentes antes de tomar una decisión. Por ejemplo, la decisión de reducir la velocidad por la detección de un posible objeto próximo, detectado en el análisis de una cámara frontal, podría no concordar con los datos obtenidos por otro componente como un *LIDAR* en el caso de un *adversarial attack*. Pero otro tipo de toma de decisiones, como por ejemplo analizar las señales de tráfico, podría ser un sistema íntegro por si mismo y tener un efecto realmente peligroso convirtiendo una señal de *Stop* en una de, por ejemplo, límite de 50 *kms/h*. Además, se pudo demostrar que varias pegatinas en una señal (ver Figura 1.5), diseñadas a modo de imitación al *graffiti*, pueden engañar a los modelos y al mismo tiempo "esconderse de la mente humana".

Figura 1.5: *Adversarial attack* en señales de tráfico

Otro ejemplo de investigación que realizó Tesla muestra como el piloto automático puede identificar un clima húmedo a través del reconocimiento de imagen y luego encender el limpiaparabrisas si es necesario. Según la investigación, con un ejemplo adverso, el sistema será corrompido y el comportamiento no será el deseado. En la Figura 1.6 se muestra la arquitectura de la red *Inception* que usan (similar a la que utilizaremos posteriormente en nuestros experimentos, ver sección 5.2.2). Está compuesta por varias capas convolucionales (ver sección 2.1) de *kernels* 3x3, 5x5 y 1x1 concatenadas y posteriormente capas de *pooling* y funciones *ReLu* (que se explicarán más adelante, ver Figura 4.2 en el capítulo 4). Los datos de entrada se corresponde a la imagen que ve la cámara frontal del vehículo, al introducirla en el clasificador *Inception*, esta imagen activa las diferentes capas resultando un valor para cada una de ellas que le lleva a un camino único y determinista para finalmente decidir si encender el parabrisas o no hacerlo. En el caso del ejemplo adverso, la imagen de entrada será a simple vista similar a la original, pero en un espacio N dimensional, el cambio será grande: el resultado de la red de la imagen original y la modificada estará lejos ya que el camino seguido en la activación de las capas es diferente.

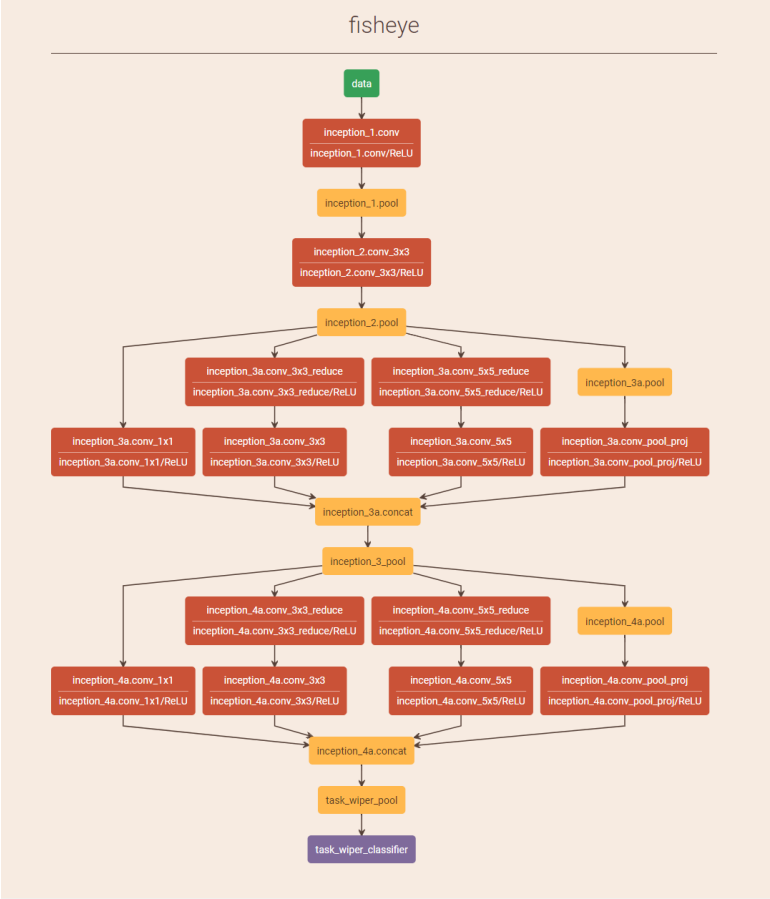


Figura 1.6: Red neuronal *Inception* detrás de los limpiaparabrisas automáticos del piloto automático Tesla

Por lo tanto, sin lugar a dudas, los *adversarial attacks* suponen una amenaza para el mundo del *deep learning* y en consecuencia para el mundo actual y futuro, sobretodo, cuando es un sistema por si mismo.

1.2 Motivación y objetivos

La mayoría de las veces, los modelos de aprendizaje automático funcionan muy bien, pero solo funcionan en una cantidad muy pequeña de todas las entradas posibles que pueden encontrar. En un espacio de alta dimensión, una perturbación muy pequeña en la entrada puede ser suficiente para causar un cambio dramático en la red neuronal. Por lo tanto, es muy fácil empujar la imagen de entrada a un punto en el espacio de alta dimensión que nuestras redes nunca hayan visto antes. Este es un punto clave a tener en cuenta: los espacios de alta dimensión son tan grandes que la mayoría de nuestros datos de entrenamiento se concentran en una región muy pequeña conocida. Este tipo de ataques, donde se construye un ejemplo

adverso, se denominan *adversarial attacks*.

En 2014, un grupo de investigadores de *Google* y de la universidad de Nueva York (NYU) descubrió que era demasiado fácil engañar a *ConvNets* con un imperceptible, pero cuidadoso cambio en la entrada [7]. El problema se ilustra en la Figura 1.7.

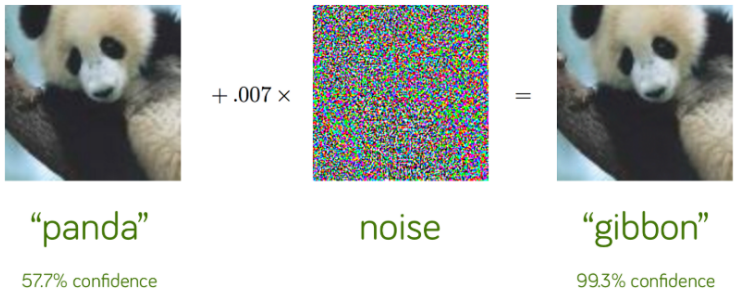


Figura 1.7: Ejemplo de *adversarial attack*

La imagen de entrada es un panda, el cual la red neuronal predice correctamente con un 57.7% de confianza. Para construir el ejemplo adverso (en este caso) se agrega un poco de ruido en el lugar correcto y la misma red neuronal ahora predice que esta imagen adversa es un gibón con un 99.3% de confianza. Para el ojo humano, el cambio es imperceptible, pero para la red no lo es. Además, hace que la red esté más “segura” de su predicción.

Por lo tanto, ¿cómo podemos saber que si una red es robusta o no? La existencia de *adversarial attacks* demuestra que una *loss* baja y un *accuray* alto, obviando ejemplos de entrenamiento solo y fijándose en imágenes de validación y test, que *a priori* miden cuanto de “bien” clasifica la red neuronal algo que nunca ha visto durante el entrenamiento (abstrayendo las características de cada clase), no es suficiente.

Una aproximación razonable es usar el propio conocimiento de que existen los ejemplos adversos para defendernos de ellos. Es decir, podemos medir cuanto de difícil es “engañar” un modelo que hasta el momento considerábamos “robusto” por sus valores de *loss* y *accuracy*.

Para engañar una red básicamente existen dos formas: ataques de caja blanca o *White-Box Attacks* y ataques de caja negra o *Black-Box Attacks*. Pero como veremos en el estado del arte, su definición no es demasiado buena. Por lo tanto, este proyecto pretende extender la actual definición para que el resultado del ataque sea más estable, preciso y fiable.

Además, vamos a cambiar la forma de seleccionar las clases objetivo del ataque para que sea más imparcial y justo. Ya que dos modelos distintos pueden ser más propensos a ser engañados por clases distintas.

Una vez que tenemos claro como queremos “engañar” al modelo, tenemos que pensar en cual es la mejor forma de hacerlo, así como la más eficaz y eficiente. Los algoritmos existentes en el estado del arte tiene dificultades para establecer balance entre cobertura y proximidad: o

buscan ser capaces de encontrar un ejemplo adverso entre cualquier entrada, aunque por ello haya que modificarla mucho; o al revés, buscan ejemplos adversos que sean muy cercanos a los datos de entrada, a costa de nos ser capaces de obtenerlos en todos los ejemplos. En virtud de este problema, nosotros vamos a proponer un nuevo algoritmo, llamado DENA: un algoritmo innovador de generación de imágenes adversas con un mejor enfoque.

¿Que ventajas ofrece DENA? Nuestro algoritmo mejora los algoritmos clásicos porque admite múltiples métricas de forma totalmente escalable. Incluye optimizadores, muy importantes a la hora de tener resultados en tiempos razonables, y sobre todo obtiene resultados fiables y portables a cualquier modelo sin depender de el, ya que el objetivo es generar imágenes que engañen a cualquier modelo. Y que funcione con cualquier tipo de *dataset* disponible.

Para demostrar que DENA es una aproximación mejor a la actual, se van a realizar una serie de pruebas contra distintos *dataset*, distintas arquitecturas de redes neuronales, varios optimizadores y funciones de coste que se explican en el apartado descripción del problema y el apartado de medida de la robustez.

En conclusión, se demostrará que las imágenes son iguales a simple vista y similares numéricamente utilizando el algoritmo contra diferentes aproximaciones para demostrar que el método presentado es mejor que las métricas actuales.

1.3 Planificación

El tiempo necesario dedicado, de forma parcial, para llevar a cabo la realización de todas las tareas del proyecto fue de 312 días laborables, dando comienzo el 25 de noviembre del 2019 y finalizando el 22 de febrero del 2021, suponiendo una media de 4 horas días, sería un total de 1248 horas de trabajo.

En este caso, se utilizó el modelo en espiral desarrollado por Barry Boehm en 1985, utilizado de forma generalizada en la ingeniería de software. Las actividades de este modelo se conforman en una espiral donde cada bucle representa un conjunto de actividades. Las actividades no están fijadas a priori, sino que las siguientes se eligen en función de la situación del bucle anterior.

En cada ciclo habrá cuatro actividades:

- Determinar o fijar objetivos
- Análisis
- Desarrollar, verificar y validar
- Análisis de resultados y planificación del siguiente ciclo

El ciclo de vida empieza en posición central. Al completar una vuelta, se obtiene un prototipo sobre el que se realizará la siguiente vuelta. De esta forma el proyecto va creciendo y asegurándose de que se entendieron los requisitos. La elección de esta metodología es debido a que permite ir modificando los modelos e incorporando nueva funcionalidad a los resultados obtenidos.

Los recursos humanos necesarios para realizar el proyecto consistieron en un analista programador y dos directores. Los recursos técnicos fueron un ordenador portátil personal en la fase de implementación y un ordenador de sobremesa proporcionado por la facultad para ejecución. Las características de ambas máquinas se enumeran en el cuadro 1.1.

Característica	Ordenador personal	Ordenador de sobremesa
Procesador	Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz	Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
Gráficos	GeForce 940MX	TITAN Xp
Tipo de SO	64 bits	64 bits
Memoria	69.6 GiB	15,5 GiB

Cuadro 1.1: Especificaciones de los ordenadores utilizados.

El sueldo bruto de un analista programador anual está fijado por convenio en las tablas salariales de 2020 y siguientes años a 24.604,37 € brutos a tiempo completo. Suponiendo unos 253 días laborables en el 2020, el coste por hora es de 12,16€. La estimación de la duración de cada tarea se puede ver en la Figura 1.8 así como la descomposición de las tareas y sus costes se puede ver en detalle en el Cuadro 1.2.

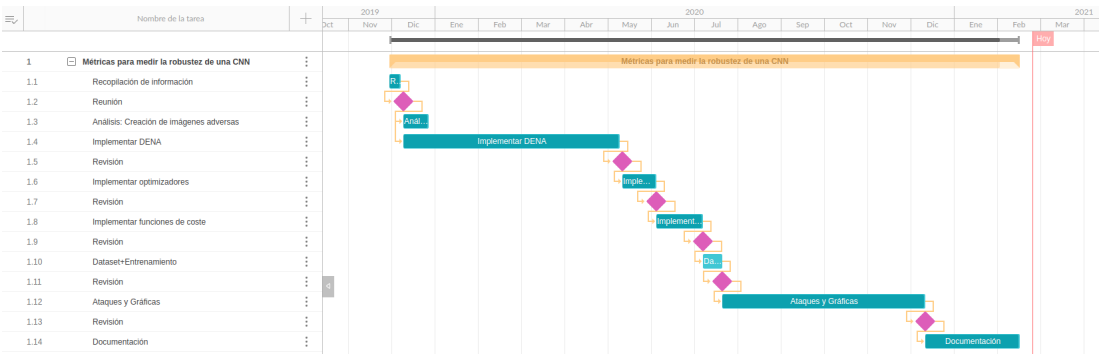


Figura 1.8: Diagrama de Gantt

En primer lugar se llevó a cabo una fase de recompilación de información y estudio de las metodologías a emplear, donde se profundiza el conocimientos e *CNN*, *adverarial attacks* y el entorno de *Keras* así como investigaciones relacionadas con el problema.

En segundo lugar se realiza la tarea de implementación del sistema y ejecución del mismo para posteriormente medir las imágenes reales contra las imágenes adversas obtenidas y poder hacer un estudio del comportamiento del sistema ante diferentes aproximaciones y poder

sacar conclusiones en base a los resultados obtenidos.

Tarea	Días	Duración (h)	Coste
Recompilación de información: Adversaria Attacks, entorno Keras	10	40	486,25 €
Reunión			
Análisis: creación de imágenes adversas	6	24	291,75 €
DENA	155	620	7.536,91 €
Implementación	80	320	3.890,02 €
Validación y pruebas	60	240	2.917,51 €
Análisis de resultados	15	60	729,38 €
Revisión			
Optimizadores	17	68	826,63 €
Recompilación de información : SGD, Momentum, Adam	10	40	486,25 €
Implementación	4	16	194,50 €
Validación y pruebas	2	8	97,25 €
Análisis de resultados	1	4	48,63 €
Revisión			
Funciones de coste + softmax	22	88	1.069,76 €
Implementación	10	40	486,25 €
Validación y pruebas	9	36	437,63 €
Análisis de resultados	3	12	145,88 €
Revisión			
Dataset y redes	15	60	729,38 €
Recopilación de información: ResNet, Inception	7	28	340,38 €
Diseño e implementación	5	20	243,13 €
Validación y pruebas	2	8	97,25 €
Análisis de resultados	1	4	48,63 €
Revisión			
Untarget Attack	30	120	1.458,76 €
Diseño e implementación	17	68	826,63 €
Validación y pruebas	10	40	486,25 €
Análisis de resultados	3	12	145,88 €
Revisión			
Target Attack	30	120	1.458,76 €
Diseño e implementación	17	68	826,63 €
Validación y pruebas	10	40	486,25 €
Análisis de resultados	3	12	145,88 €
Revisión			
Gráficas	10	40	486,25 €
Diseño e implementación	5	20	243,13 €
Validación y pruebas	3	12	145,88 €
Análisis de resultados	2	8	97,25 €
Revisión			
Documentación	30	120	1.458,76 €
Revisión			
Total	325,00	1300,00	15.803,20 €

Cuadro 1.2: Desglose de tareas y costes del proyecto

1.4 Herramientas software

El código se desarrolla en *Python* debido a la variedad de módulos y algoritmos que proporciona, además de que posee una curva de aprendizaje rápida para el usuario. Se usan librerías como *Tensorflow*, *Keras* y *NumPy* que facilita el aprendizaje en las primeras fases.

1.4.1 Tensorflow

TensorFlow es una interfaz para expresar algoritmos de *machine learning* proporcionando una implementación para ejecutarlos. Un cálculo expresado con *TensorFlow* se puede ejecutar con poco o ningún cambio en una amplia variedad de sistemas heterogéneos, desde dispositivos móviles como teléfonos y tabletas hasta sistemas distribuidos a gran escala de cientos de máquinas y miles de dispositivos computacionales como tarjetas GPU . El sistema es flexible y se puede utilizar para expresar una amplia variedad de algoritmos, incluidos algoritmos de entrenamiento e inferencia para modelos de *deep neural networks*. Se ha utilizado para realizar investigaciones y desplegar sistemas de aprendizaje automático en producción en más de una docena de áreas de ciencias de la computación y otros campos, incluidos el reconocimiento de voz, la visión por computadora, la robótica, la recuperación de información, el procesamiento del lenguaje natural, la extracción de información geográfica ..., entre otros.

La implementación de la interfaz ha sido creada por el equipo de *Google Brain Team* dentro de la organización de investigación de inteligencia de máquinas de *Google*. La API de *TensorFlow* [8] y una implementación de referencia se lanzaron como un paquete de código abierto bajo la licencia Apache 2.0 en noviembre de 2015.

TensorFlow es una herramienta para manejar sistemas capaces de construir y entrenar redes neuronales, para detectar y descifrar patrones y correlaciones. Análogos al aprendizaje y razonamiento usados por los humanos. El nombre *TensorFlow* deriva de las operaciones que tales redes neuronales realizan sobre arrays multidimensionales de datos. Estos arrays multidimensionales son referidos como"tensores".

Al escribir un programa *TensorFlow*, el objeto principal que manipula y pasa es el tensor, representando un cálculo parcialmente definido que eventualmente producirá un valor. Primero se construye un gráfico a partir del programa, detallando cómo se calcula cada tensor en función de los otros tensores disponibles y luego se ejecutan partes de este gráfico para lograr los resultados deseados.

Tensorflow también puede calcular automáticamente los gradientes del modelo. Esto es posible ya que las expresiones matemáticas de cada neuronal del modelo se guardan e el gráfico, por lo que el gradiente de todo el gráfico se puede calcular utilizando la regla de la cadena para el cálculo de las derivadas al optimizar la función de coste. Esta característica es muy útil tanto en tiempo de entrenamiento como a la hora de escribir código, evitando posibles errores.

1.4.2 Keras

Keras es una API [9] de redes neuronales de alto nivel, escrita en *Python* pensada inicialmente para ser capaz de ejecutarse sobre *TensorFlow*, *Microsoft Cognitive Toolkit* (CNTK) o

Theano. Actualmente, *Keras* está integrada dentro de *Tensoflow*, de forma que sea más fácil de usar sin sacrificar la flexibilidad y el rendimiento.

Keras fue diseñada para facilitarle el uso a los seres humanos, reduciendo la carga cognitiva: ofrece API consistentes y simples, minimiza la cantidad de acciones del usuario requeridas para casos de uso comunes, y proporciona comentarios claros y procesables sobre el error del usuario.

Keras presenta módulos independientes totalmente configurables que se pueden conectar con la menor cantidad de restricciones posible. En particular, las capas neurales, las funciones de coste, los optimizadores, los esquemas de inicialización, las funciones de activación y los esquemas de regularización son módulos independientes que puede combinar para crear nuevos modelos. Los nuevos módulos son fáciles de agregar (como nuevas clases y funciones), y los módulos existentes proporcionan amplios ejemplos. Poder crear fácilmente nuevos módulos permite una expresividad total, lo que hace que *Keras* sea adecuado para la investigación.

1.4.3 NumPy

NumPy es el paquete de *open source* de *Python*, que le agrega mayor soporte para vectores y matrices, constituyendo una API [10] de funciones matemáticas de alto nivel para operar con esos vectores o matrices.

1.5 Estructura de la memoria

A lo largo de esta memoria, se profundizará en los siguientes aspectos:

- Introducción: En el primer capítulo se hace una breve introducción sobre *adversarial attacks* y como afectan a la fiabilidad de una red neuronal, siendo estos los motivos que lleva a trabajar en este proyecto. Además, se refleja la viabilidad del mismo, el posible impacto, la planificación que hemos seguido y se describen brevemente las herramientas utilizadas.

- Descripción el problema: En este capítulo se profundiza en el conocimiento de las redes neuronales y porque su fiabilidad puede verse comprometida por ataques adversos. Se estudian las técnicas de ataques clásicos que se han realizado hasta el momento y finalmente se presenta DENA, un nuevo algoritmo que mejora los proyectos anteriores y se concreta en sus diferentes componentes.

- Medidas de robustez: En este capítulo se explica como podemos asegurarnos de la robustez de una red neuronal convolucional con la nueva propuesta, quedando constancia de que es una metodología más fiable que las aproximaciones extendidas hasta el momento.

- Resultados: A lo largo de este capítulo se presentarán los diferentes experimentos que hemos realizado, las estrategias utilizadas, las mejoras en la velocidad de cómputo y los resultados obtenidos.
- Conclusiones: Finalmente, se exponen los aspectos más relevantes a extraer de la realización del proyecto, así como posibles líneas de trabajo futuro.

Capítulo 2

Descripción del Problema

En este capítulo se explican las distintas capas que forman parte de una CNN, se explica por qué es posible engañarlas mediante las imágenes adversas y cómo generarlas.

2.1 Redes Convolucionales

Una red convolucional o CNN (por sus siglas en inglés) es un tipo de red neuronal artificial de aprendizaje supervisado, es decir, transforma los datos de entrada en una salida esperada. Trata de imitar el córtex visual del ojo humano para identificar objetos y «ver», con una estructura que está formada por unas entradas, que llamaremos ejemplos de entrenamiento; varias capas ocultas especializadas y ordenadas jerárquicamente de forma que las primeras capas pueden detectar líneas, curvas y se van especializando hasta llegar a capas más profundas que reconocen formas complejas como un rostro o una silueta; y por último, una serie de operaciones (funciones de activación), pesos y conexiones entre capas que harán funcionar la red de forma que cada configuración sea adaptable al tipo de problema que intentemos resolver.

Hoy en día, las CNN son usadas para resolver un amplio rango de problemas, entre ellas destaca la clasificación de imágenes (en el que nos centraremos), detección de objetos, problemas de regresión, segmentación, conducción autónoma, aprendizaje por refuerzo por ejemplo en videojuegos, rumbas, robótica, etc.

2.1.1 Capa de entrada

La red toma como entrada los píxeles de una imagen realizando, posteriormente, una asignación de los píxeles de la misma a las neuronas, por ejemplo, una imagen con apenas 28 x 28 píxeles de alto y ancho, equivaldría a 789 neuronas, en una escala de grises (1 color). Si tuviéramos una imagen a color, necesitaríamos tres canales (rojo, verde y azul) y la asignación sería de 28 x 28 x 3 = 2,352 neuronas de entrada. El número de parámetros crece de manera

significativa por lo que será muy importante tener en cuenta la optimización a lo largo de la construcción de la red.

El valor de cada píxel se normaliza a un rango [0-1]. La normalización de los datos generalmente acelera el aprendizaje y conduce a una convergencia más rápida. Si las imágenes tienen una característica que es totalmente positiva o totalmente negativa, esto hará que el aprendizaje sea más difícil para los nodos de la capa siguiente. La imagen al ir atravesando los pesos de las neuronas en las diferentes capas tendrán que hacer *zigzag* como los que siguen en una función de activación sigmoidea (ver sección 2.1.3). La normalización asegura que la magnitud de los valores que asume una característica sean más o menos iguales.

2.1.2 Capas de convolución

El proceso distintivo entre una red neuronal y una red convolucional es el proceso de convolución cuyo propósito es extraer las características de una imagen. Como tal, una convolución consiste en tomar grupo de píxeles cercanos de la imagen de entrada e ir operando matemáticamente contra una pequeña matriz a la que se le denomina *kernel*. Este recorre todas las neuronas de entrada - de izquierda a derecha y de arriba hacia abajo - y genera una nueva matriz de salida, que será la nueva capa de neuronas ocultas, y que también se conoce como matriz de activación.

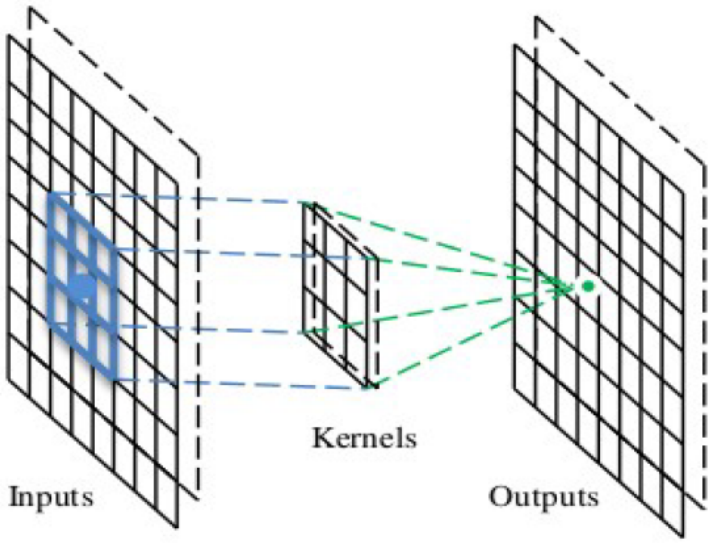


Figura 2.1: Operación de convolución

En realidad, no aplicamos un solo *kernel*, sino que tendremos un conjunto de ellos. Por ejemplo si se escogen 32 filtros, se obtienen un total de 32 matrices de salida (a este conjunto se le denomina *feature mapping*), siguiendo con el ejemplo anterior, cada salida de 28 x 28 x

3, multiplicado por 32, da un total de 75.264 neuronas para nuestra primera capa oculta. En esta primera convolución, se obtendrían 32 imágenes nuevas con ciertas características de la imagen original y que permitirían hacer una clasificación futura.

Siguiendo la analogía de las neuronas del córtex visual, estas poseen unos campos receptivos que responden a estímulos localizados en una determinada zona. Las neuronas de cada capa no están totalmente conectadas con todas la neuronas de la siguiente capa, sino solo con un subconjunto de ellas. Este hecho nos permitirá reducir el cómputo.

2.1.3 Funciones de activación

A medida que vamos obteniendo la imagen filtrada por el *kernel*, se aplica una función de activación buscando funciones cuyas derivadas sean simples para minimizar así el coste computacional.

Las funciones de activación (o de transferencia), determinan la salida de una capa de la red neuronal. Son usadas por cada una de las neuronas que conforman la red para determinar si deben de activarse o no según si la información que reciben es relevante para la predicción del modelo. Cada neurona tiene un peso, y multiplicando ese peso por la entrada (o *input*) de esa neurona obtenemos la salida de la misma, que es transferida a la siguiente capa.

Para que cada neurona se active en función de la relevancia se utiliza la técnica de ***back-propagation***: método de cálculo del gradiente hacia atrás que propaga los errores desde las capas de salida hasta las capas de entrada para mejorar el valor de los pesos de las interconexiones entre las capas al mismo tiempo que se irán ajustando los pesos de cada neurona hasta ser óptimos.

El sistema recibe una imagen, esta pasa por cada capa oculta de la red realizando las operaciones necesarias hasta que se genera una salida. Como hemos mencionado antes, CNN es aprendizaje supervisado, es decir, tenemos una etiqueta (salida deseada) con la cual comparamos la salida. El error generado (diferencia o distancia entre ambos valores) se propaga hacia las capas de entrada a todas las neuronas que contribuyen directamente a dicha salida y de forma proporcional a su contribución.

La característica principal de las funciones de activación es que deben ser derivables y computacionalmente eficientes, ya que estos resultados son computados en millones de neuronas no solo al calcular la salida para cada neurona con ellas, sino también la influencia que ésta ejerce en la salida y en la modificación de pesos en consecuencia.

Entre las funciones de activación se destacan las siguientes (ver Figura 2.2):

- **Rectified Linear Units (ReLU)** Esta función se aplica a la salida de cada neurona de forma que todas las activaciones negativas (no relevantes) pasan a 0. Su propósito es agregar no linealidad al modelo, eliminando la relación proporcional entre la entrada

y salida. Permiten que la red entrene más rápido, factor de vital importancia ante una gran cantidad de datos.

$$f(x) = \max(0, x)$$

- **Softplus** Es una aproximación suavizada de la función ReLU.

$$f(x) = \ln(1 + e^x)$$

- **Sigmoide** Normaliza los números de entrada en un rango [0,1] donde valores altos se les asigna 1 y valores muy bajos se les asigna 0. Este tipo de activación tiene una convergencia lenta además de saturar (pesos muy altos, baja tasa de aprendizaje) y matar el gradiente (multiplicar por 0) al acotarla a un rango reducido, por lo cual se suele utilizar en la última capa: valores muy altos dan una alta probabilidad de pertenencia a la clase de salida.

$$\phi(z) = \frac{1}{1 + e^{-x}}$$

- **Softmax** Es una generalización multiclase de la función sigmoidea, de modo que la suma de todas las neuronas de salida sea igual a la unidad. Se considera esta función como una distribución de probabilidad categórica, asignando un grado de confianza a cada clase de salida.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

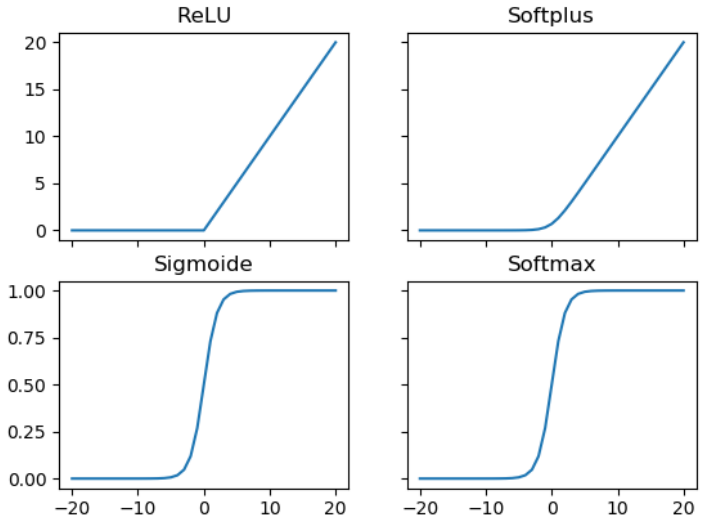


Figura 2.2: Diferentes tipos de activación

Centrándonos en el problema de "*hacking*", aquí se encuentra la razón de porque es tan difícil defender las redes neuronales. Aunque nuestras redes neuronales son no lineales por definición, la función de activación más común que usamos para entrenarlas es *ReLu* por su estabilidad y rapidez en entrenamiento. Pero eso también hace posible impulsar la función de activación a valores arbitrariamente altos. Al observar esta compensación entre la capacidad de entrenamiento y la solidez de los ataques adversos, podemos concluir que los modelos de redes neuronales que hemos estado utilizando son intrínsecamente defectuosos. La facilidad de optimización ha tenido el coste de modelos que se pueden engañar fácilmente.

El otro parámetro clave que hace que los modelos sean vulnerables a ataques es el concepto propagación del gradiente. Con el, el entrenamiento funciona muy bien porque mínimos cambios en la entrada pueden magnificar y cambiar su comportamiento, haciéndolas más robustas, pero tiene el efecto colateral de que si la entrada tiene ruido, se propagan características que no son correctas. Y no es trivial detectar que valores son incorrectos ni dónde se activan para eliminarlos.

2.1.4 Capas de pooling

Si a partir de la primera convolución, en la que teníamos 75.264 neuronas realizáramos una segunda convolución el número de neuronas crecería exponencialmente implicando mayor procesamiento.

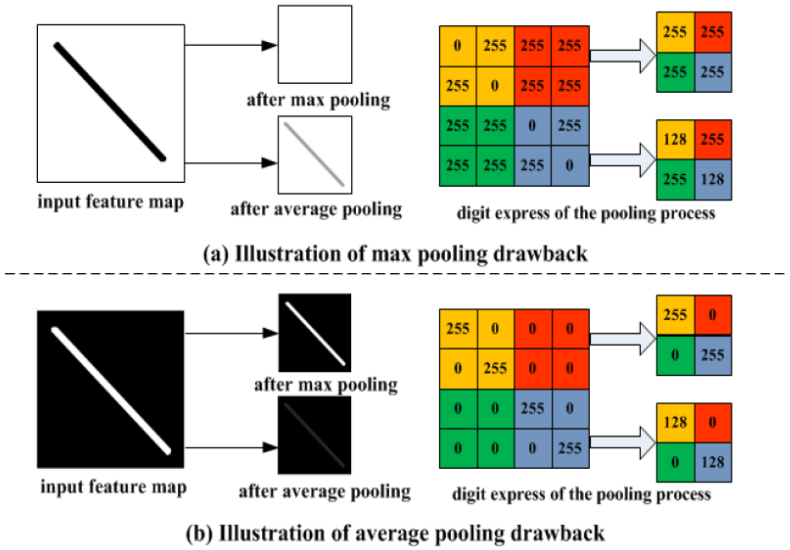


Figura 2.3: Resultado de aplicar diferentes tipos de pooling

El objetivo de estas capas es disminuir la carga computacional del sistema y al mismo tiempo ayudar con la caracterización de la imagen obteniendo y localizando los rasgos predo-

minantes en ella. A partir de cada *feature mapping*, calculamos la media (*average-pooling*) o el máximo valor (*max-pooling*) de una región de la imagen, de esta forma la matriz resultante tendrá unas dimensiones considerablemente menores que las obtenidas en capas convolucionales, como se puede ver en la Figura 2.3.

2.1.5 Capas de dropout

Dropout es un método que desactiva un número determinado de neuronas en una red neuronal de forma aleatoria. Las neuronas desactivadas no se tienen en cuenta para la propagación hacia delante ni hacia atrás, lo que obliga a las neuronas cercanas a no depender tanto de las neuronas desactivadas. La red aprende que hay ocasiones en las que algunas de las características se activan más que otras, pero seguirán perteneciendo a la misma clase, cuando siga presente alguna de ellas. Esto es muy importante, puesto que, como comentábamos, es inabarcable entrenar recreando todas las situaciones posibles, así como las múltiples combinaciones entre los propios atributos del objeto en cuestión.

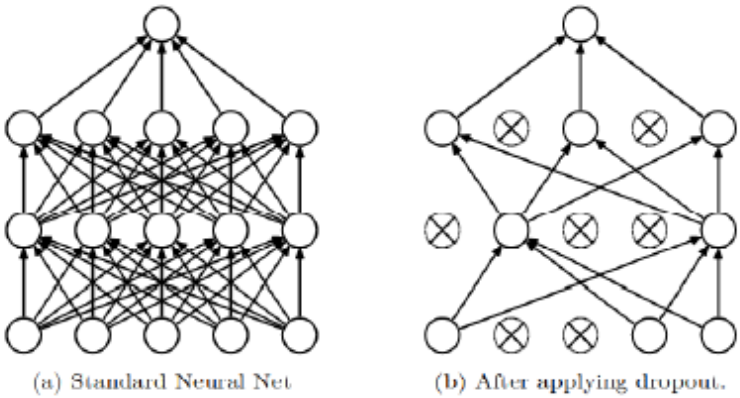


Figura 2.4: Diferencia entre red sin dropout (a) y la misma red aplicando dropout (b)

2.1.6 Capa completamente conectada

Se trata de la última capa de la CNN que realiza la clasificación basándose en las características extraídas por las capas de convolución y reducidas por *pooling*. En esta capa todos los nodos están conectados con la capa precedente a diferencia de las capas de convolución.

Está compuesta por un número de neuronas igual al número de clases en las que es posible clasificar una entrada. La salida de esta capa, será un vector donde cada componente represente la probabilidad de pertenencia a la clase resultado. Se determinará cual de estos valores es mayor y se dará una salida al clasificador.

2.2 Ataques adversos

Como se comenta anteriormente, las redes neuronales son muy precisas, pero se ha demostrado que pueden ser engañadas mediante ataques adversarios: pequeñas modificaciones en los datos de entrada con el objetivo de engañar al sistema. Estos ejemplos adversos no dependen mucho de la red neuronal profunda específica utilizada para la tarea. Un ejemplo adverso entrenado para una red parece confundir también a otra. En otras palabras, varios clasificadores asignan la misma clase (incorrecta) a un ejemplo adverso. Basándose en esto, en este apartado se estudian los algoritmos clásicos de ataque y se explica porqué no es completa su definición. Finalmente se aporta una nueva solución.

2.2.1 Algoritmos de ataque clásicos

Se pueden clasificar los ataques según el conocimiento que se tiene de modelo que se va a atacar: *Black Box Attacks* o modelos de caja negra y *White Box Attacks* o modelos de caja blanca. Opcionalmente podemos hacer una cuarta división intermedia entre los dos, dando lugar a los *Gray Box Attacks* o modelos de caja gris.

En el modelo de ***Black Box Attack***, un adversario no conoce la estructura de la red objetivo o los parámetros, pero puede interactuar con el algoritmo de *Deep Learning* para consultar las predicciones de entradas específicas. Los atacantes siempre elaboran muestras adversas en un clasificador sustituto entrenado por los pares de datos: entrada y predicción adquiridos de muestras benignas.

Debido a la transferibilidad de las muestras adversas, los ataques de caja negra siempre son independientes del modelo de partida y pueden comprometer a otro modelo.

En el modelo de ***Gray Box Attack***, un atacante conoce la arquitectura del modelo de destino, pero no tiene acceso a los pesos de la red. El atacante también puede interactuar con el algoritmo de *Deep Learning*.

En este modelo de amenaza, el atacante elabora muestras adversas en un clasificador sustituto con la misma arquitectura. Debido a la información adicional de la estructura, un ataque de caja gris siempre muestra un mejor rendimiento en comparación con un ataque de caja negra.

Los ***White Box Attacks*** son los ataques más fuertes, es decir, el atacante tiene acceso completo al modelo objetivo, incluidos todos los parámetros, lo que significa que el atacante puede adaptar los ataques y crear directamente muestras adversas en el modelo objetivo. Estos son los tipos de ataque que consideraremos en este proyecto.

Los ataques más exitosos son los métodos basados en gradientes. Es decir, los atacantes modifican la imagen en la dirección del gradiente de la función de pérdida con respecto a la imagen de entrada. Hay dos enfoques principales para realizar tales ataques:

- **One-shot attacks**: en los que el atacante da un solo paso en la dirección del gradiente.
- **Iterative attacks**: donde en lugar de un solo paso, se itera sobre un mismo ejemplo hasta conseguir un ejemplo adverso válido.

Los métodos de un solo paso tienen tasas de éxito más bajas en comparación con los métodos iterativos en los ataques de caja blanca, sin embargo cuando se trata de ataques de caja negra, los métodos de un solo paso resultan ser más efectivos. La explicación más probable de esto es que los métodos iterativos tienden a sobreajustarse a un modelo en particular.

A continuación se exponen los métodos más comunes en la generación de imágenes adversas con ataques de tipo *White Box Attacks*:

Método de signo de gradiente rápido (FGSM)

Este método [7] calcula la imagen adversa agregando una perturbación de la magnitud del píxel en la dirección del gradiente. Esta modificación se calcula en un solo paso, por lo que es muy eficiente en términos de tiempo de cálculo.

$$x_{adv} = x + \epsilon \leq \mathbf{sign}((\nabla_x J(x, y_{true}))$$

donde:

- x el la imagen original de entrada, x_{adv} es la imagen adversaria, J es la función de coste y y_{true} es la clase para la salida x.

Método de señal de gradiente rápido dirigido (T-FGSM)

De manera similar a FGSM, este método [11] se calcula en un paso de gradiente, pero en este caso en la dirección del gradiente negativo con respecto a la clase objetivo:

$$x_{adv} = x \quad \epsilon \leq \mathbf{sign}(\nabla_x J(x, y_{target}))$$

donde:

- y_{target} es la clase que es seleccionada como objetivo para el ataque adverso.

Método de signo de gradiente rápido iterativo (I-FGSM)

Los métodos iterativos toman T pasos de gradiente de magnitud $\alpha = \frac{\epsilon}{T}$ en lugar de un solo paso t.

$$x_0^{adv} = x$$

$$x_{t+1}^{adv} = x_t^{adv} + \alpha \leq \mathbf{sign}(\nabla_x J(x_t^{adv}, y))$$

Centrándonos en ataque de caja blanca (los realizados en el proyecto), podemos hacer otra clasificación de los ataques según la clase *target* elegida para confundir la red:

- ***Non-targeted Adversarial Attack***: el objetivo del ataque no dirigido es modificar ligeramente la imagen original para que sea clasificada incorrectamente por el modelo sin importarnos cual es la clase resultado.

- ***Targeted Adversarial Attack***: el objetivo del ataque dirigido es modificar ligeramente la imagen original de manera que la imagen se clasifique como una clase de destino especificada y el modelo de aprendizaje automático desconocido clasifique la imagen de entrada como clase *target*.

En 2017, una de las competiciones del NIPS [12], un congreso cuyo propósito es fomentar el intercambio de investigaciones, trató la temática de este trabajo titulando la competencia como *Adversarial Attacks and Defences* y planteando problemas de seguridad en sistemas de aprendizaje automático para acelerar la investigación sobre ejemplos adversarios y la solidez de los clasificadores de aprendizaje automático. Esta competición plantea tres subtipos: *Non-targeted Adversarial Attack*, *Targeted Adversarial Attack*, explicados en la sección 2.2.1, y añade un subtipo llamado *Defense Against Adversarial Attack* cuyo objetivo de la defensa es construir un clasificador de aprendizaje automático que sea robusto al ejemplo adversario, es decir, que pueda clasificar las imágenes adversarias correctamente.

En cada una de estas subcompeticiones se invita a realizar y presentar un programa que resuelva la tarea correspondiente. Al final de la competición se aplican todos los ataques contra todas las defensas para evaluar cómo se comporta cada uno de los ataques contra cada una de las defensas.

Boosting Adversarial attacks with Momentum (MI-FGSM) [13] fue el ataque ganador en *Non-targeted Adversarial Attack* y *Targeted Adversarial Attack*. Este método utiliza el impulso para mejorar el rendimiento de los métodos de gradiente iterativo.

Input: The logits of K classifiers $\boldsymbol{l}_1, \boldsymbol{l}_2, ..., \boldsymbol{l}_K$; ensemble weights $w_1, w_2, ..., w_K$; a real example $\boldsymbol{x}$ and ground-truth label y;
Input: The size of perturbation ϵ; iterations T and decay factor μ.
Output: An adversarial example $\boldsymbol{x}^*$ with $\|\boldsymbol{x}^* - \boldsymbol{x}\|_\infty \leq \epsilon$.

$\alpha = \epsilon/T$;
$\boldsymbol{g}_0 = 0$; $\boldsymbol{x}_0^* = \boldsymbol{x}$;
for $t = 0$ to $T - 1$ **do**
  Input $\boldsymbol{x}_t^*$ and output $\boldsymbol{l}_k(\boldsymbol{x}_t^*)$ for $k = 1, 2, ..., K$;
  Fuse the logits as $\boldsymbol{l}(\boldsymbol{x}_t^*) = \sum_{k=1}^{K} w_k \boldsymbol{l}_k(\boldsymbol{x}_t^*)$;
  Get softmax cross-entropy loss $J(\boldsymbol{x}_t^*, y)$ based on $\boldsymbol{l}(\boldsymbol{x}_t^*)$ and Eq. (9);
  Obtain the gradient $\nabla_{\boldsymbol{x}} J(\boldsymbol{x}_t^*, y)$;
  Update $\boldsymbol{g}_{t+1}$ by Eq. (6);
  Update $\boldsymbol{x}_{t+1}^*$ by Eq. (7);
end for
return $\boldsymbol{x}^* = \boldsymbol{x}_T^*$.

Figura 2.5: Pseudo-código MI-FGSM NIPS 2017

En la Figura 2.5 se muestra el pseudo-código de MI-FGSM el cual, utiliza los gradientes de los pasos t anteriores con un decaimiento μ y el gradiente del paso t+1 para actualizar la imagen del adversario en el paso t+1. Los resultados muestran que este método supera a todos los demás métodos de la competencia y muestra buenos resultados de transferibilidad, es decir, funciona bien en *Black Box Attacks*.

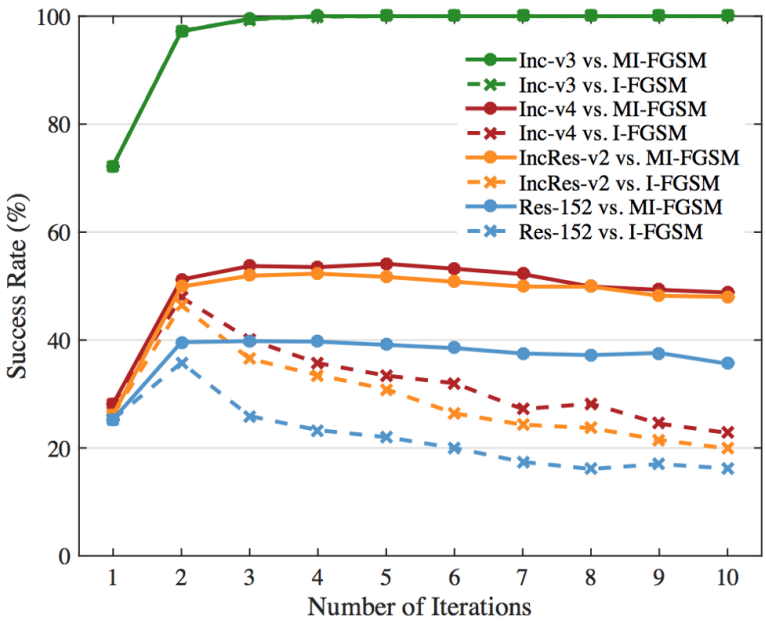


Figura 2.6: Resultados NIPS 2017

En el gráfico de la Figura 2.6, se muestra la tasa de éxito frente al ńumero de iteraciones. Todos los ataques que se muestran se generaron utilizando el modelo *Inception-V3*. Los mo-

delos objetivo son el propio *Inc-V3* (*White Box Attack*) y otras cuatro redes, *Inc-v4*, *IncRes-V2* y *Res-152* (*Black Box Attacks*).

Los resultados muestran que el enfoque basado en el impulso logra el mismo nivel de éxito en los ataques de caja blanca y el método iterativo básico, pero supera sistemáticamente al segundo método en todos los ataques de caja negra. Está claro que el rendimiento de los ataques de caja negra disminuye muy rápidamente con el número de iteraciones en el método iterativo básico, mientras que aumenta en el método basado en impulso para un mayor número de iteraciones, e incluso, después de eso, la disminución es muy lenta.

La defensa ganadora en la subcompetición *Defense Against Adversarial Attack* fue *The High level representation guided denoiser* [14]. Esta solución se basa en la observación de que, a pesar de que las perturbaciones adversas son bastante pequeñas a nivel de píxel, se amplifican en toda la red, produciendo un ataque adversario.

Algunos resultados oficiales de NISP sobre *Adversarial learning* pueden encontrarse en: 2017 [15], 2018 [16].

Como se puede observar, los métodos clásicos tienen ciertos problemas ya que: no son capaces de encontrar un ejemplo adverso siempre, la clase *target* es cualquier clase que engañe el modelo y esta no es la mejor opción porque puede ser que haya clases más fáciles de engañar que otras. Además no se tiene en cuenta la cantidad de píxeles a modificar ni el porcentaje de cambio.

2.3 Aproximación propuesta: DENA

Para crear un método de generación de imágenes adversas más robusto a los anteriores se propone una nueva función llamada *Delayed Elastic Net Attack* (DENA). Dena es el algortimo clave para conseguir evaluar bien la robustez y capacidad de geralización de una CNN, a continuación se explica su funcionamiento: Con un ejemplo x y un modelo f, el proceso de obtención de un ejemplo adverso es similar al *Elastic-Net attack* propuesto en [17], con estas variantes:

- **Delay**: Las regularizaciones l_1 y l_2 (controlados por los parámetros λ_1 y λ_2, respectivamente) no se introducen desde el principio ya que son muy difíciles de controlar y puede suponer no conseguir encontrar el ejemplo adverso. Los introducimos cuando ya hemos obtenido una imagen que confunde la red, es decir, $L_{gen} = 0$. De esta forma, principalmente somos capaces de encontrar un ejemplo adverso, para posteriormente usar la regularización con el objetivo de seguir encontrando más ejemplos, pero que sean más cercanos a los datos de entrada. Funciona de manera similar a como lo haría el aire que hincha un globo, primero se expande hasta encontrar los límites (en nuestro

caso hasta encontrar ejemplos adversos) y una vez encontrados, se empieza a encoger hasta el punto de partida (el ejemplo más cercano al original dentro de los adversos).

- **Búsqueda del ejemplo más cercano**: En lugar de utilizar el algoritmo de umbral de contracción (ISTA) [18, 19], para reducir el número de características a modificar, cuando aparezca un ejemplo adverso, se usarán los pesos y se observará si se ha cambiado la orientación. En el caso de que se haya producido un cambio de negativo a positivo, se establece una diferencia a cero.

Algorithm 1 Delayed Elastic Net Attack (DENA).

procedure DENA($x, f, L_{gen}, extraIters, lr, \lambda_1, \lambda_2, f_{dist}$)
  $t \leftarrow 0$;
  $x_{adv} \leftarrow x$;
  $d \leftarrow \infty$;
  $x_{best} \leftarrow x$;
  while $t < extraIters$ **do**
    $y_{pred} \leftarrow f(x_{adv})$;
    $L \leftarrow L_{gen}(y_{pred})$;
    $g \leftarrow \frac{\partial L}{\partial x}$;
    if $L = 0$ **and** $f_{dist}(x, x_{adv}) < d$ **then**
      $d \leftarrow f_{dist}(x, x_{adv})$;
      $x_{best} \leftarrow x_{adv}$;
    end if
    if $L = 0$ **then**
      $g \leftarrow g + \lambda_1 \left\|x - x_{adv}\right\|_1 + \frac{\lambda_2}{2} \left\|x - x_{adv}\right\|_2^2$;
    end if
    if $t > 0$ **or** $L = 0$ **then**
      $t \leftarrow t + 1$;
      $s_p \leftarrow sing(x - x_{adv})$;
      $x_{adv} \leftarrow x_{adv} - lr * g$;
      $s_a = sing(x - x_{adv})$;
      $x_{adv} \leftarrow where(s_p * s_a \leq 0, x, x_{adv})$;
    else
      $x_{adv} \leftarrow x_{adv} - lr * g$;
    end if
  end while
  return x_{best};
end procedure

El pseudocódigo 1 muestra el pseudocódigo del algoritmo *DENA* cuyo significado de variables se muestra a continuación:

- x es la imagen de entrada.
- f es el modelo CNN previamente entrenado, con entradas x o similares con el mismo conjunto de salida.
- L_{gen} es la función de coste aplicada al modelo para obtener L.

- $extraIters$ es el número máximo de iteraciones una vez encontrado x_{adv}, para asegurar robustez a la nueva imagen modificada.
- lr es el *Learning Rate* aplicado a la imagen, regula el nivel de cambio de x_{adv}.
- λ_1 y λ_2 parámetros de regularización y cambio de g.
- f_{dist} es la función de distancia entre x y x_{adv}.
- t cuenta el número de veces que un x_{adv} pasa por la fase de colapso.
- x_{adv} es el ejemplo adverso resultado de aplicar iterativamente el algoritmo *DENA*. El x modificado de acuerdo a f_{dist} y L_{gen}.
- x_{best} es el mejor candidato de x_{adv} hasta el momento, el mejor se calcula en base a f_{dist} (la mínima distancia) y L_{gen} (ejemplo adverso válido L=0).
- y_{pred} es la predicción del modelo para la entrada especificada (x_{adv}).
- L es el *loss* del modelo, generada a partir de L_{gen}.
- g es el gradiente del modelo calculado a partir del L y la imagen de entrada x.
- d última mejor (menor) distancia entre x y x_{adv} (distancia entre x_{best} y x).
- s_p y s_a aplican la fase de colapso, restringe el mecanismo de colapso, evitando que la distancia entre x y x_{adv} cambie su orientación.

El proceso se lleva a cabo en dos fases:

- **Fase de expansión**: Mientras que no se produzca un ejemplo adverso, la solución se propaga lejos del ejemplo original.
- **Fase de colapso**: Cuando se obtiene un ejemplo adverso, forzamos a que el ejemplo sea lo más similar posible a la solución real.

Capítulo 3

Medidas de robustez

Hasta el momento, para medir la robustez se usaban de las funciones de pérdida pero estas tienen poco sentido si luego se demuestra que son fáciles de engañar. En esta sección se va un paso más allá en la definición de que es "engañar" a un modelo en la definición de tipos de ataques adversos y como controlar las modificaciones que se están realizando.

A continuación, se explica como gracias a las funciones de pérdida, usándolas a nuestro favor, logramos obtener ejemplos adversos cercanos a las imágenes originales para finalmente, medir el cambio que se realizaron en ellas en detalle con las funciones de distancia.

3.1 Variaciones a los tipos de ataques adversarios

Como mencionamos en la sección 2.2.1, la elección de como engañar al modelo no es todo lo robusta posible. En nuestro caso, variamos esta definición para aumentar la confianza del ataque.

- ***Untargeted attacks***: la definición clásica menciona que no importa cual es la clase resultado, pero esta aproximación podría sesgar las clases objetivo hacia una clase en concreto que fuera "más sencilla" de engañar. En nuestro caso, tenemos control sobre la clase *target* que se elija además de la confianza depositada en ella, exigiendo que haya n clases más probables que la clase real.

- ***Target attacks***: en la definición clásica veíamos que sí se sabe *a priori* cual es la clase *target*, pero de nuevo, esta definición es muy abstracta ya que no tenemos ningún tipo de control sobre la cantidad de modificaciones que hay que realizarle a una imagen para que sea adversa. Además, no se habla en ningún momento de la confianza que tiene el modelo en la clase objetivo, en nuestro caso, obligamos a que cada imagen adversa ese lo más cercana posible a las imágenes originales convirtiéndolo en un problema de minimización, ver sección 3.1 y que el modelo esté un porcentaje segura de que

realmente es la clase ganadora.

3.2 Funciones de pérdida

Cuando se empieza a entrenar una red neuronal, inicializamos lo pesos al azar, y obviamente esto no dará muy buenos resultados. En el proceso de entrenamiento, queremos comenzar con una red neuronal de bajo rendimiento y terminar con una red con alta precisión. Este proceso lo llevan a cabo las funciones de pérdida (conocidas como *loss function*): se encargan de cuantificar cuanto se está equivocando la predicción de la red neuronal respecto a la clase *target*. Queremos que la función de pérdida sea mucho más baja al final del entrenamiento. Es posible mejorar la red, porque podemos cambiar su función ajustando pesos y encontrar otra función que de mejores resultados que la inicial.

En el proceso de generación de ejemplos adversos, lo que buscamos es que esta función de pérdida mantenga al modelo con una buena precisión ante imágenes modificadas. Por lo tanto, en el algoritmo DENA, se usan las funciones de pérdida para modificar los datos de entrada y engañar el modelo (ver sección 3.1), cambiándole la clase *target*.

Para encontrar estos ejemplos adversos, dado que queremos que sean lo más parecidos a los datos de entrada, el problema se convierte en un problema de minimización con las restricciones mencionadas en el apartado anterior.

Intentamos minimizar la distancia del ejemplo adverso con los datos de la imagen de partida, pero con las restricciones que tiene cada tipo de ataque y así estar seguros de la robusted de los datos.

El problema de clasificación implica asignar a cada ejemplo una probabilidad de certeza a cada clase del conjunto de clases disponibles. Para nuestro caso, de construir ejemplos adversos, necesitamos minimizar la función de pérdida para que la clase original cambie su probabilidad de certeza a otra clase. A continuación se describe el enfoque para construir ejemplos adversos. Se empieza utilizando el algoritmo DENA (ver algoritmo 1 en el capítulo 2) que define formalmente el problema de encontrar un ejemplo adverso para una imagen x como sigue:

$$\begin{aligned} &\text{minimize } D(x, x+\delta) \\ &\text{such that } C(x+\delta) = t \\ &\qquad x+\delta \ / \ [0,1]^n \end{aligned} \tag{3.1}$$

Dado un valor fijo de la variable x, el objetivo es encontrar la minimización de la función de distancia $D(x, x+\delta)$. Es decir, queremos encontrar qué pequeño cambio tenemos que hacer en una imagen x para que el clasificador cambie su salida y que el resultado siga siendo

una imagen válida.

3.2.1 Funciones objetivo

El problema reflejado en las ecuaciones 3.1 es difícil de formular directamente con los algoritmos existentes, ya que la restricción $C(x+\delta) = t$ es altamente no lineal, por lo que se realiza una nueva expresión de la misma, más adecuada para la optimización: definimos una función objetivo f tal que:

$$C(x+\delta) = t \infty f(x+\delta) \Leftarrow 0$$

Existen muchas posibilidades para f, pero no todas tienen la misma finalidad: unos objetivos están centrados en ***untargeted attacks*** y otros para ***targeted attacks***, nos centraremos en:

- ***untargeted attacks***

$$\begin{aligned} f_2(x') &= \sum_{i=0}^{n} y^N_{target} \leq \mathbf{relu}(\max(y_{true} \leq x') \quad x') \\ f_3(x') &= \sum_{i=0}^{n} y^N_{target} \leq \mathbf{softplus}(\max(y_{true} \leq x') \quad x') \quad \log(2) \end{aligned} \tag{3.2}$$

- ***targeted attacks***

$$\begin{aligned} categorical\_cross\_entropy(x') &= \sum_{i=0}^{n} \quad y^N_{target} \leq \log(\mathbf{clip}(x', \epsilon, 1)) \\ use\_thresh\_value(x') &= \sum_{i=0}^{n} y^N_{target} \leq \mathbf{relu}(thresh \quad x') \end{aligned} \tag{3.3}$$

donde:

- x' es la imagen de la cual se calcula la función de coste.
- y^N_{target} es un vector binario, con unos en la clase objetivo, 0 en el resto.
- y_{true} es la clasificación correcta (***one-hot endoced label***).
- $clip$ recorta la entrada a los límites con valores ***min***, ***max***.
- $thresh$ es el limite variable de la función de coste ***use_thresh_value***.

Algunas de las fórmulas añaden una constante (si la función sigue respetando la definición). Esto no impacta sobre el resultado final, ya que sólo escala la función de minimización.

En lugar de formular el problema como:

$$\begin{aligned} &\text{minimize } D(x, x+\delta) \\ &\text{such that } f(x+\delta) \Leftarrow 0 \\ &\qquad\qquad x+\delta \ / \ [0,1]^n \end{aligned} \tag{3.4}$$

se usa esta otra formulación alternativa:

$$\begin{aligned} &\text{minimize } D(x, x+\delta) + c * f(x+\delta) \\ &\text{such that } x+\delta \ / \ [0,1]^n \end{aligned} \tag{3.5}$$

Ambas ecuaciones son equivalentes, en el sentido de que existe un $c > 0$ tal que la solución óptima para este último coincide con la solución óptima para la primera.

3.3 Funciones de distancia

Es necesario cuantificar cuanto se diferencian la imagen de entrada y la imagen resultado del *Adversarial Attack*, y así elegir solo la imagen más similar a la original. Para medir la diferencia se usan medidas de distancia entre píxeles, en concreto, en este trabajo usaremos 4 medidas de distancia: $norma_1$ o distancia de Manhattan, $norma_2$ o distancia euclídea, $norma_0$ y la $norma_\infty$.

La $norma_1$ y la $norma_2$ nos da una idea de cuanto de media se están modificando los píxeles y la distancia entre ellos. Pero no sabemos realmente si se modifican todos los píxeles o si se modifican solo un subconjunto de la imagen, además no sabemos en que ratio cambia cada píxel. Para salir de dudas, podemos usar la $norma_0$ que nos da una idea del porcentaje de píxeles que hay que modificar de media. y la $norma_\infty$, que multiplicada por 100 nos da una idea del porcentaje que se varió en los píxeles modificados sobre el máximo que se está a hacer para engañar al modelo. A continuación se muestran las fórmulas:

1. La norma 0:
$$\sum_{i=0}^{n} x_i \ \Rightarrow\!\!\!\!\!\!\Leftrightarrow \ x_i \models 0$$
2. La norma 1:
$$l1 = \sqrt{x_1 \quad y_1} + \sqrt{x_2 \quad y_2} + .. + \sqrt{x_N \quad y_N}$$
3. La norma 2:
$$L2 = \sqrt{(x_1 \quad y_1)^2 + (x_2 \quad y_2)^2 + ... + (x_N \quad y_N)^2}$$

4. La norma $\in$:

$$L_{\infty} = \max(\sqrt{x_1}, \sqrt{x_2}, \ldots, \sqrt{x_N}) = \max_{i \epsilon 1,\ldots,N} \sqrt{x_i}$$

El algoritmo *DENA* acepta el uso de múltiples métricas de forma escalable, y para cada una de las métricas calcula el ejemplo mínimo.

Capítulo 4

Experimentación

En este capítulo se muestran los experimentos realizados en cada una de las aproximaciones, explicando cada una de las técnicas que fueron utilizadas y los cambios y mejoras que se fueron incorporando durante el proceso.

4.1 Entrenar los modelos

Se eligieron modelos *ResNet* e *Inception* porque son redes comúnmente usadas y testeadas. Ambos modelos, sobretodo *Inception* son modelos demasiado grandes para resolver el problema de clasificación de *MNIST* y *CIFAR10* por lo cual se adaptaron ambos modelos a soluciones más sencillas como se muestra en las Figuras 4.1 y 4.2. En las Figuras se muestra *ResNet* con el *dataset MNIST* e *Inception* con el *dataset CIFAR10*. En ambos modelos es necesario cambiar el *'input'* del modelo ya que cada imagen de *MNIST* tiene un tamaño de 28 x 28 x 1 y de *CIFAR10* un tamaño de 32 x 32 x 3.

Las imágenes de entrada al modelo están normalizadas en el rango 0-1 (ver 2.1.1), ya que durante del descenso de gradiente (ver sección 2.1.3), la velocidad de aprendizaje es proporcional a la magnitud de las entradas.

Si las entradas son de diferentes escalas, los pesos conectados a algunas entradas se actualizaran mucho más rápido que otras. Esto generalmente perjudica el proceso de aprendizaje, a menos que sepamos de antemano qué características son más importantes que otras, en cuyo caso podemos ajustar las escalas para que la red neuronal enfoque su aprendizaje en lo más importante. Pero en la práctica es bastante improbable que podamos o sepamos que características beneficiarán al aprendizaje de antemano.

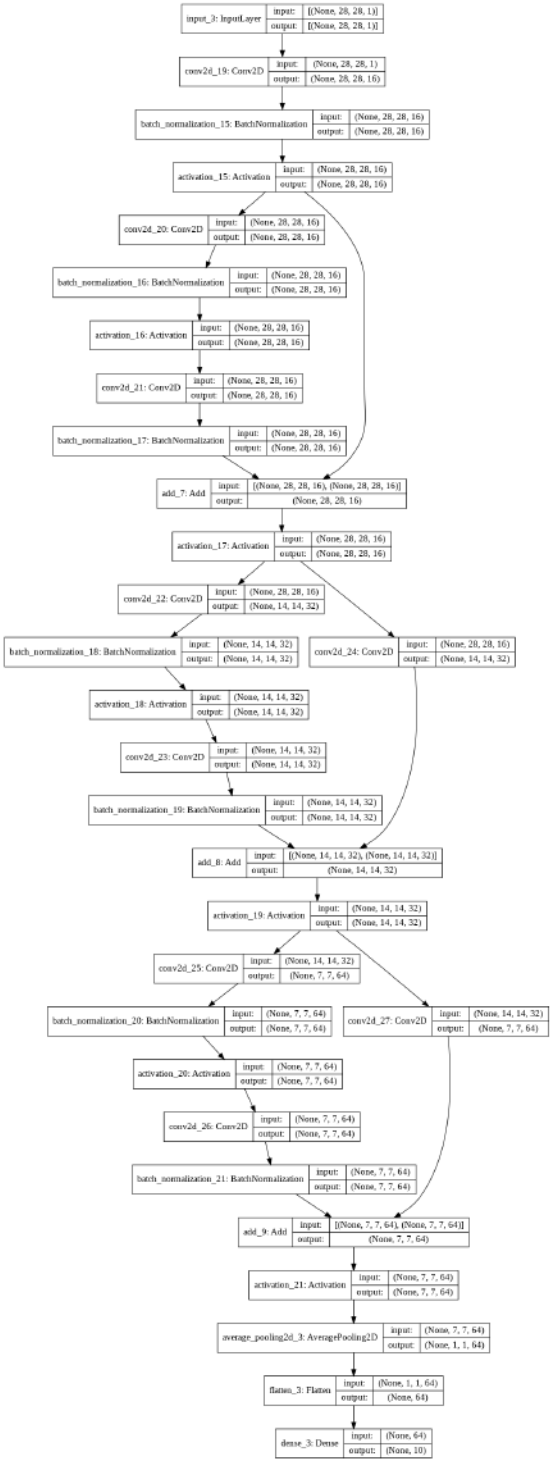


Figura 4.1: Arquitectura del modelo *ResNet* usado con *MNIST*

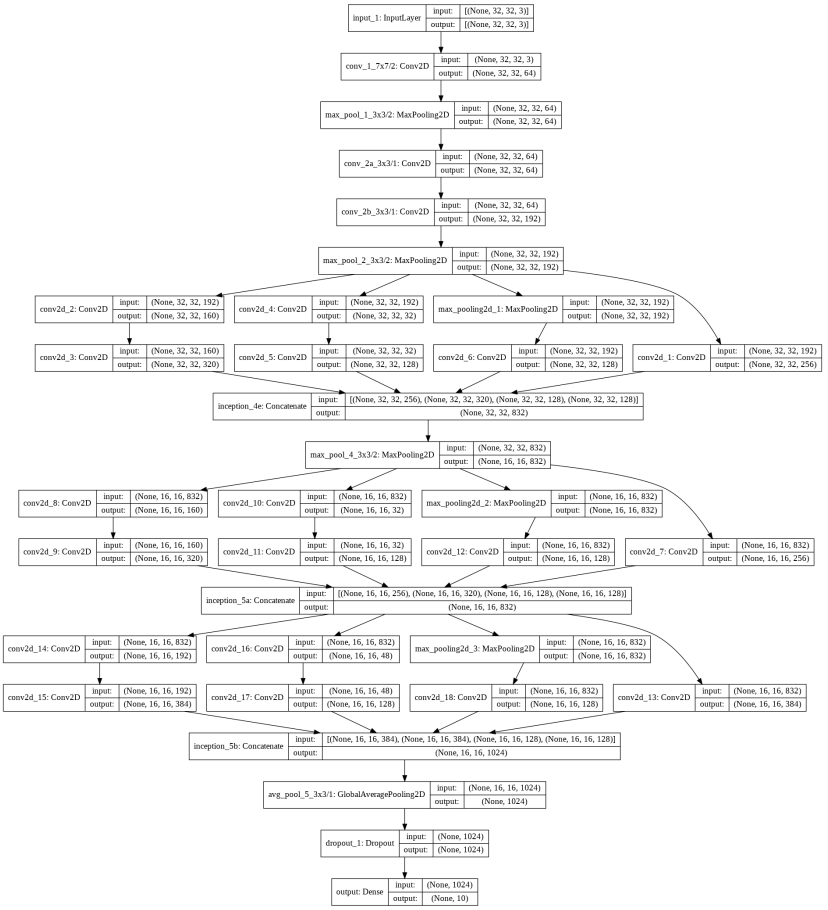


Figura 4.2: Arquitectura del modelo *Inception* usado con *CIFAR10*

4.2 Implementación de los ataques

Se realizaron los dos tipos de ataques vistos: *untarget attacks* y *targeted attacks* (ver sección 2.2.1) y mejorados en la sección 3.1, los cuales tienen ciertas características diferentes a la hora de realizar la implementación y obtener los resultados deseados cambiando la definición de que es "engañar" a un modelo.

4.2.1 Características de implementación de ataques untargeted

- **Funciones de coste usadas**: *f2* y *f3* (ver sección 3.2).

- El ***target*** (ver algoritmo 4.2.3) calculado en cada iteración del bucle principal de Dena (ver algoritmo 1). No nos interesa que gane una clase en particular, sino que el objetivo es que gane cualquier otra clase que no sea la verdadera, el problema es que cuando una imagen se modifica para incrementar la probabilidad de pertenencia a una clase especifica, puede que también se incremente la probabilidad de pertenencia a otras clases. Así que las etiquetas objetivo son actualizadas en cada paso.

- **Aceptación de ejemplo adverso**: un ejemplo se considera válido cuando el modelo predice que la clase resultado de la imagen (no clase verdadera) está en términos de probabilidad por encima de las n clases. De forma que la clase verdadera tendrá al menos n clases más probables a ser candidatas para el resultado.

4.2.2 Características de implementación de ataques target

- **Funciones de coste usadas**: *categorical_cross_entropy* y *use_thesh_value* (ver sección 3.3).

- El ***target*** (ver algoritmo 4.2.3) se calcula una sola vez, la primera vez que se introduce la imagen antes de modificarla. Esto es debido a que solo queremos que gane una clase en particular y durante todas las modificaciones, el gradiente será desplazado hacia la clase que en términos de probabilidad, es la clase n más probable de ser la ganadora.

- **Aceptación de ejemplo adverso**: se introduce un nuevo parámetro, el parámetro $thesh$. Un ejemplo adverso se considerará válido cuando la confianza en la clase ganadora esté por encima de un *threshold*. Es decir, el modelo esté $thesh\%$ seguro de que la clase ganadora es la clase obtenida de la función *getTarget* (ver 4.2.3) que se explica a continuación.

4.2.3 Obtención del *target*

La clase *target* es la clase que queremos que sea la ganadora y se puede calcular de varias formas dependiendo si se trata de un *untarget attack* o un *targeted attack*. El objetivo es que la clase "real" no sea la clase ganadora. En nuestro caso esta función se calcula de la siguiente forma:

```
1    def getTargetN(n, y_true, y_pred, attack_type):
2        # a que clase pertenece cada ejemplo
3        y_pred_test_classes = np.argmax(y_true, axis=1)
4        output = np.zeros_like(y_pred)
5        example = np.arange(len(y_true))
6        # -1 a la clase que pertenece
7        y_pred[example, y_pred_test_classes] = -1.
8        if attack_type == 'untargeted':
9            # 1 a los n mejores (excluyendo la clase verdadera)
10           clase1 = np.argsort(-y_pred, axis=1)[:, :n]
11           output[np.expand_dims(example, axis=1), clase1] = 1.
12       elif attack_type == 'target':
13           # 1 a la clase n con más prob (excluyendo la clase
     verdadera)
14           clase1 = np.argsort(-y_pred, axis=1)[:, n - 1]
15           output[example, clase1] = 1
16       return output
```

Donde:

- Tanto la salida como la entrada es un vector con el tamaño del número de clases, en formato *one_hot_encode* con 1 en la clase target y 0 la clase verdadera, se pone a -1 en la línea 7 para evitar que la clase real interfiera en el cálculo de las clases objetivo.
- En el caso de que $attack\_type == ``untargeted"$ la n significa que queremos que haya por como mínimo N clases erróneas con una probabilidad más alta que la clase verdadera.
- En el caso de que $attack\_type == ``target"$ el significado de n varía. Se le asigna 1 a aquella clase con la clase N de la predicción real con más probabilidad de ser la ganadora.

4.3 Algoritmos de optimización

Los algoritmos de optimización definen el ritmo de aprendizaje de la red. Dentro del proceso de cuantificar el error que lleva acabo el *backpropagation* corrigiendo los pesos de cada

neurona mediante la regla de la cadena. Existe otro elemento llamado **optimizador**, que se encarga de evaluar cuánto y cómo corregir el peso a lo largo del aprendizaje.

Debemos encontrar el conjunto de pesos que minimice la función de pérdida, es decir, encontrar el mínimo de la función. Con la derivada (de aquí la importancia de que las funciones de activación fueran derivables que se explica en la sección 2.1.3 del capítulo 2) se obtiene la pendiente en un punto por lo cual encadenando dichos puntos, se llega a un mínimo: óptimamente el mínimo global. El problema está cuando se estanca en un mínimo local, por esto, hay distintos tipos de optimizadores que se adaptarán mejor o peor a nuestra función de coste: *SGD*, *RMSprop*, *Adagrad*, *Adadelta*, *Adam*, *Adamax*, *Nadam*, etc. En este trabajo nos centraremos en: **Stochastic Gradient Descent (SGD)**, **Momentum** y **Adam**.

4.3.1 *Stochastic Gradient Descent (SGD)*

El gradiente es la generalización vectorial de la derivada. Es un vector de tantas dimensiones como la función y cada dimensión contiene la derivada parcial en dicha dimensión:

$$f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots \frac{\partial f}{\partial x_n}\right]$$

El gradiente f_x es el vector que contiene la información de cuanto crece la función en un punto especifico x por cada dimensión de nuestra función de forma independiente. Entre esta información se encuentra la dirección de aumento más pronunciado de la función. Como queremos minimizar, daremos un paso en la dirección opuesta del gradiente. Si calculamos el gradiente de la función de pérdida con los pesos (valores a cambiar respecto a las cuales tomar las derivadas) y en dirección opuesta del gradiente, nuestra función de pérdida disminuirá gradualmente hasta que converja. Este algoritmo es el conocido como *Gradient Descent*

$$w_j = w_j \quad lr\partial\frac{L}{\partial w_j}$$

Para cada peso w, se resta la derivada con respecto a él por la tasa de aprendizaje (o *learning rate*), lr. La tasa de aprendizaje controla la importancia que le damos a cada iteración. Es uno de los parámetros más importantes de redes neuronales ya que si se elige un valor demasiado grande, puede llegar a "saltarse" el mínimo. Es decir, el algoritmo no llegará a converger. Por el contrario, al elegir una tasa de aprendizaje demasiado pequeña, el algoritmo tardará mucho tiempo en converger.

Para calcular las derivadas de la pérdida, deberíamos analizar cada ejemplo de nuestro conjunto de datos, acción que resulta poco eficiente para calcular, en cada, iteración, la pendiente del gradiente, porque en cada iteración solo se mejora un pequeño paso. Para resolver este problema, hay otro algoritmo, *Batch Gradient Descent o Mini-Batch Gradient Descent* don-

de la regla para actualizar los pesos sigue siendo la misma, pero no se calcula una derivada exacta, si no que, aproximamos la derivada para *batch* y usaremos esa derivada para actualizar los pesos, aunque no se garantiza que tome la dirección óptima. De hecho, generalmente no lo hará. Si se elige una tasa de aprendizaje suficientemente baja, no se garantiza que la función de pérdida disminuya (como en *Gradient Descent* simple) sino que su pérdida disminuirá con el tiempo, pero fluctuará y será mucho más "ruidosa". El tamaño de *batch* a utilizar para estimar las derivadas es otro parámetro a elegir. Por lo general, lo deseable sería que fuera de un tamaño de *batch* tan grande como pueda manejar su memoria.

La versión extrema del *Batch Gradient Descent* con un tamaño de *batch* igual a 1 se llama *Stochastic Gradient Descent*. En la literatura moderna y muy comúnmente, cuando leemos *Stochastic Gradient Descent (SGD)*, en realidad se refieren al *Batch Gradient Descent*.

4.3.2 *Momentum*

El *Momentum*, también conocido como *SGD con momentum o SGD con impulso* es un método que ayuda a acelerar los vectores de gradientes en las direcciones correctas, lo que conduce a una convergencia más rápida.

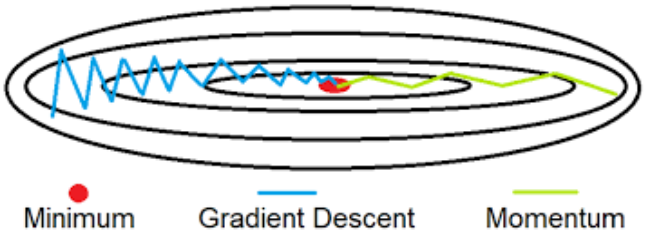


Figura 4.3: Comparativa entre: SGD y Momentum

El SGD tiene problemas en áreas donde la superficie tiene una curva mucho más abrupta en una dimensión que en otra (regiones comunes alrededor de óptimos locales). En estos escenarios, SGD avanza vacilante hacia el óptimo local como el trazo azul de la Figura 4.3. Imaginemos una pelota rodando por una colina, esta se acercará a un mínimo antes o después pero a medida que pasa el tiempo avanza más rápido, con más aceleración. Para añadir este componente de tiempo, añadimos el parámetro del impulso que toma valores entre 0 y 1. Este aumenta para las dimensiones cuyos gradientes apuntan en las mismas direcciones y reduce las actualizaciones para las dimensiones cuyos gradientes cambian de dirección. Como resultado, ganamos una convergencia más rápida y una oscilación reducida como en el trazo verde de la Figura 4.3.

$$V_t = \beta v_{t-1} + (1 \quad \beta)g$$

Se calcula un promedio ponderado de una secuencia de datos *V*. Cada nueva actualización, dependerá de la anterior ponderado por el impulso β. El resto de la fórmula es igual que la anterior donde de cada secuencia importan lo últimos $(1 \quad \beta)$ puntos de secuencia por la derivada del gradiente.

4.3.3 *Adam*

Adam usa *Momentum* (4.3.2) y tasas de aprendizaje adaptativo (*Adaptive Learning Rate*) para converger más rápido. Las tasas de aprendizaje adaptativo parten de la idea de que empezamos con pasos grandes y terminamos con pasos pequeños. Esto nos permite movernos más rápido inicialmente y a medida que disminuye la tasa de aprendizaje, tomamos pasos más pequeños. Lo que permite que la red converja más rápido ya que no sobrepasamos el mínimo con pasos grandes.

$$\theta_{t+1} = \theta_t \quad \frac{\eta * m_t}{\sqrt{v_t} + \epsilon}$$

donde

$$m_t = \beta_1 m_{t-1} + (1 \quad \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 \quad \beta_2) g_t^2$$

- ϵ impide la división por cero..
- η es la tasa de aprendizaje. En Keras se iguala a $\eta = 0.001$.
- β_1 y β_2 son términos de decaimiento. Generalmente $\beta_1 = 0.9$ y $\beta_2 = 0.999$

4.4 Implementación de DENA

Lo primero que se hizo fue adaptar el algoritmo 1 para que en lugar de ejecutar una imagen de cada vez, aceptase un *batch* de imágenes en paralelo. De forma que cada imagen del *batch* tiene un número diferente de iteraciones del bucle principal.

Para ello se llama a DENA una sola vez de la siguiente forma

$$DENA(x, f, L_{gen}, extraIters, lr, \lambda_1, \lambda_2, f_{dist}, batchSize)$$

Donde x contiene todo el *dataset*. Es necesario llevar la cuenta de que posiciones están siendo procesadas, cuales están en espera de procesarse y el orden para que los ejemplos adversos resultado se puedan comparar con las imágenes de entrada.

La segunda modificación fue implementar 3 optimizadores: (ver secciones: SGD 4.3.1, Momentum 4.3.2 y Adam 4.3.3). Para acelerar la modificación de la imagen, quedando las líneas

22 y 24:

$$x_{adv}[i] = opt.step(i, x_{adv}[i], g[i])$$

y Dena:

$$DENA(x, f, L_{gen}, extraIters, lr, \lambda_1, \lambda_2, f_{dist}, batchSize, opt)$$

donde :

- $x_{adv}[i]$ es una imagen que cumple la condición.
- i es el índice o índices dentro de la lista que cumplen la condición.
- *opt* es una clase que implementa el optimizador (inicializado en el exterior del algoritmo).
- *.step* es la propia implementación de la fórmula a cada paso.
- g es el gradiente.

Hay que tener en cuenta que las clases que implementan los optimizadores y usan datos de la iteración anterior necesitan de un mecanismo de borrado cuando la imagen es válida como ejemplo adverso y se pasa a procesar otra en su misma posición. En el cual deberán inicializarse los valores del optimizador a los valores por defecto iniciales.

La tercera modificación fue introducir un parámetro para acelerar el cómputo, en concreto, aumentar el parámetro de lr del optimizador en uso pasadas n iteraciones sin encontrar ni un solo ejemplo adverso. Se inicializa fuera del bucle el contador $eproch\_witchout\_find\_adv$ con tantas posiciones como el tamaño del batch y en la línea 24, se va sumando uno a uno cada posición que cumpla la condición. Si el contador llega a n iteraciones en una posición, el lr de esa imagen en concreto aumentará de la siguiente forma:

```
1  DENA(...){
2      ...
3      eproch_witchout_find_adv = np.zeros(batch_size, dtype=int)
4      ...
5          xAdv[i] = opt.step(i, xAdv[i], g[i])
6          eproch_witchout_find_adv[i] += 1
7          idx = np.argwhere(eproch_witchout_find_adv[i] % 500 == 0)
8          opt.lr_list[i[idx]] = opt.lr_list[i[idx]] * 5
9      ...
10 }
```

Se usó un incremento de $lr_t = lr_{t-1} \leq 5$.

Todo el código trabaja con números decimales con precisión de 32 bits. Es importante mantener esta precisión en cada variable para evitar desbordamientos o números indeseados como *NaN* o $\in$.

En el caso de las funciones de pérdida, para calcular el *loss* y el gradiente. Se sobreescribe la función *softmax* 3.2.1, para evitar *NaN* en el *loss* y en consecuencia en el gradiente.

```
1  def custom_softmax(x, alfa=15, epsilon=1):
2      alfa = np.float32(alfa)
3      epsilon = np.float32(epsilon)
4      x_max = K.max(x, axis=1, keepdims=True)
5      x_min = K.min(x, axis=1, keepdims=True)
6      n = alfa * (x - x_max)
7      d = K.stop_gradient(K.maximum(alfa, x_max - x_min + epsilon))
8      x_exp = K.exp(n / d)
9      custom_activation = x_exp / K.sum(x_exp, axis=1, keepdims=True)
10
11     return custom_activation
```

El último cambio que se hizo del algoritmo original de DENA (ver algoritmo 1) fue cambiar la forma de detectar que el modelo "se equivoca". Es decir, hacer el cambio de $L = 0$ por una condición equivalente con la finalidad de eliminar los problemas de precisión y $\in$. La nueva condición implica que *DENA* conozca la clase verdadera y comprueba que no es la que se predice en la línea 7.

4.4.1 Uso de las funciones de coste

Para usar las funciones de coste (ver sección 3.2.1) y obtener la *loss* del modelo de acuerdo a una entrada, así como su gradiente, la librería de *Keras* proporciona una función *K.function()* para implementar y modificar parámetros de un modelo ya entrenado.

```
tf.keras.backend.function(
    inputs, outputs, updates=None, name=None, **kwargs
)
```

Donde:

- *inputs*: una lista de tensores de entrada.
- *outputs*: una lista de tensores de salida.
- *updates*: una lista de opciones para la actualización.
- *name*: nombre de una función.
- ***kwargs*: que son argumentos usados por el *K.Session.run()*

Como salida, se obtiene una lista de valores.

En resumen, recibe valores, computa en tensores y devuelve valores de una función definida que utiliza el API de *Keras*.

En nuestro código, la función *get_loss_grad_attack* define la función *K.function* con la función de coste a usar. Se define en Dena antes del bucle principal y se llama en cada iteración del bucle de la siguiente forma:

```
1     def DENA:
2         ...
3         targets = keras.layers.Input(shape=model.output_shape[1:])
4         lf = K.learning_phase()
5         get_loss_grad_attack = K.function(
6             [model.inputs[0], model.targets[0], targets, lf],
7             custom_adv_loss_grad(attack_type, loss_fun, thresh,
    model.outputs[0], model.targets[0], targets, model)
8         )
9         while (~inserted).any():
10            yPred = model.predict(xAdv)
11            yTarget = getTargetN(n, y_active, yPred.copy(),
    attack_type)
12            info = get_loss_grad_attack([xAdv, y_active, yTarget,
    0])
13            g = info[1]
14        ...
15
```

Es importante destacar el parámetro ***kwargs*, correspondiente a la línea 4 de 4.4.1 en la definición, que debe ir a 0 a la hora de llamar a la función *get_loss_grad_attack* para indicar que no se va a modificar nada como en la línea 12.

4.5 Optimizaciones de código

La optimización del algoritmo es la parte clave de la implementación, ya que evitará esperas eternas a la hora de obtener resultados. Entre las optimizaciones usadas se encuentran:

- Eliminación de bucles *for*, excesivamente lentos en *Python*. Se hace uso del *broadcasting* y *list compression*.

- Reducción del número de imágenes para generar ejemplos adversos. Debido al tiempo de ejecución, finalmente se entrenó con las imágenes de entrenamiento haciendo división de este subconjunto en un 80% entrenamiento y un 20% test. Las imágenes de test de los propios *dataset* fueron usadas para generar ejemplos adversos.

- Una de las líneas de código que resulta en cuello de botella para el resto de operaciones

es la función *predict* que nos da el resultado de la predicción del modelo de la entrada pasada por parámetro. Si usamos la función *K.function*, que puede devolver más de un valor, en lugar de hacer el *predict* del modelo en cada iteración fuera y en la propia función, hacemos que *K.function* devuelva también el *model.outputs[0]* de forma evitamos calcularlo dos veces y la línea 5 del algoritmo 4.4.1 quedaría:

```
1        get_loss_grad_attack = K.function(
2        [model.inputs[0], model.targets[0], targets, lf],
3
    custom_adv_loss_grad(attack_type,loss_fun,thresh,model.outputs[0],
    model.targets[0], targets, model),
4        (model.outputs[0], )
5
```

y la línea 10 del algoritmo 4.4.1 solo se haría 1 vez por cada ejemplo (la primera vez) y el resto de las veces se haría:

$$yPred = info[2]$$

- Introducción de un parámetro para regular el número de iteraciones que se intentaba obtener un ejemplo adverso sin conseguirlo. De forma que si $eproch\_witchout\_find\_adv > 5000$ iteraciones para un ejemplo, se considera que no se ha conseguido encontrar ese ejemplo adverso y se descarta de los resultados. Cabe destacar, que el algoritmo *Dena* (ver algoritmo 1) siempre encuentra ejemplos adversos, solo hay que tener tiempo necesario para que este se encuentre. "Encontrar" significa hacer las iteraciones suficientes para agregar la cantidad de ruido en la imagen de forma imperceptible y que se cambie la clase resultado.

4.6 Inicialización

Para cada una de las iteraciones, hay valores de inicialización que no van a variar a lo largo de los experimentos. Para escoger su valor, se hizo un análisis de cada parámetro para buscar la configuración óptima y balanceada en aspectos de calidad (distancias mínimas) como en velocidad (tiempo de ejecución). Este estudio está detallado en la sección 5.3, y los parámetros escogidos son:

- $batch\_size = 100$
- $lr = 1^{-3}$ para los optimizadores SGD y Momentum y $lr = 1^{-4}$ para el optimizador Adam.
- $\lambda_1 = 0.01$

- $\lambda_2 = 0.01$
- $extraIters = 250$
- $\beta = 0.9$: solo para el optimizador Momentum.
- $\beta_1 = 0.9$: solo para el optimizador Adam.
- $\beta_2 = 0.999$: solo para el optimizador Adam.
- $\epsilon = 10^{-8}$: solo para el optimizador Adam.

4.7 Ejecuciones y resultados

Se realizaron pruebas con la configuración del cuadro 4.1.

			Loss function	
Dataset	Architecture	Opt	Untarget attack	Target attack
MNIST	Resnet	SGD	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
		Momentum	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
		Adam	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
	Inception	SGD	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
		Momentum	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
		Adam	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
CIFAR	Resnet	SGD	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
		Momentum	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
		Adam	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
	Inception	SGD	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
		Momentum	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)
		Adam	f2	categorical_cross_entropy (CE)
			f3	use_thesh_value (TV)

Cuadro 4.1: Ejecuciones realizadas para *untarget attacks* y *target attacks*.

Capítulo 5

Resultados

A lo largo de este capítulo se presentarán los resultados obtenidos del resultado obtenidos acorde a los datos del capítulo de 4. La primera aproximación abarca los resultados de los ataques de tipo *untargeted* y la segunda aproximación se corresponde con los ataque de tipo *targeted*.

Se parte de los dos modelos entrenados, con la arquitectura citada en la sección 5.2 y a partir de ahí, sin modificar el modelo, se ejecuta cada una de las aproximaciones donde se miden las distancias entre las imágenes verdaderas y las imágenes adversas con las normas vistas en las sección 3.3.

5.1 Conjunto de datos

En este proyecto haremos uso de bases de datos muy conocidas en el ámbito de clasificación de imágenes con CNN: *MNIST*, *Fashion-MNIST*, *CIFAR10* y *CIFAR100*.

5.1.1 MNIST

La base de datos *MNIST* (ver Figura 5.1) está formada por imágenes de dígitos escritos a mano [20].

Contiene 60,000 ejemplos de entrenamiento y un conjunto de prueba de 10,000 ejemplos de aproximadamente 250 personas que aportaron su escritura. Los dígitos han sido normalizados por tamaño y centrados en una imagen en blanco y negro (un canal) de tamaño fijo de 28x28. Los píxeles se organizan en filas (almacenados en una matriz C[1]). Los valores de píxel son de 0 a 255. Donde 0 significa fondo (blanco) y 255 significa objeto (negro). Los valores de las etiquetas son del 0 al 9, correspondiéndose con el número que refleja la imágen.

MNIST es un subconjunto de un conjunto más grande disponible llamado *NIST*. En el artículo [21], utilizando un sistema jerárquico de CNN, logra obtener una tasa de error en la

[1]Es decir, el índice de la última dimensión cambia más rápido.

base de datos *MNIST* de 0.23%.

Figura 5.1: Subconjunto MNIST

5.1.2 Fashion-MNIST

El conjunto de datos *Fashion-MNIST* [22] es un conjunto de datos de las imágenes de artículos de *Zalando* [23], que consta de 60.000 ejemplos de entrenamiento y un conjunto de prueba de 10.000 ejemplos. Cada ejemplo es una imagen en escala de grises de 28x28, asociada con una etiqueta. *Fashion-MNIST* se construyo con la idea de servir de reemplazo directo del conjunto de datos *MNIST* original (ver sección 5.1.1) para la evaluación comparativa de los algoritmos de aprendizaje automático. A continuación se muestra un ejemplo de las imágenes de este *dataset* (ver Figura 5.2):

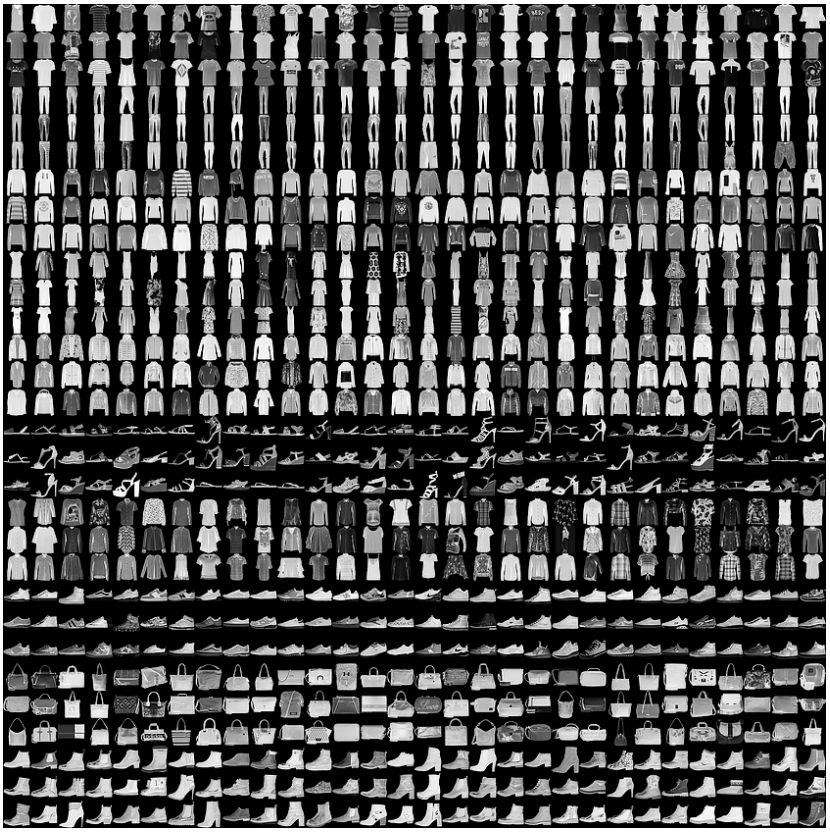


Figura 5.2: Subconjunto Fashion-MNIST

Cada imagen tiene un tamaño de 28x28x1 y está asociada a una etiqueta de temática ropa. En todo el conjunto de datos, hay 10 clases, al igual que *MNIST* (ver sección 5.1.1). Por ejemplo:

$$img[0] == \text{“}Sneaker\text{”}$$

$$img[1] == \text{“}Trouser\text{”}$$

$$\dots$$

5.1.3 CIFAR-10 y CIFAR-100

Existen dos conjuntos de datos *CIFAR*: El *CIFAR-10* y el *CIFAR-100* [24].

El conjunto de datos *CIFAR-10* consta de 60,000 imágenes en color de 32x32 en 10 clases, con 6,000 imágenes por clase. Hay 50,000 imágenes de entrenamiento y 10,000 imágenes de prueba.

El conjunto de datos se divide en cinco lotes de entrenamiento y un lote de prueba, cada uno con 10,000 imágenes. El lote de prueba contiene exactamente 1,000 imágenes seleccionadas al azar de cada clase. Los lotes de entrenamiento contienen las imágenes restantes en

orden aleatorio, pero algunos lotes de entrenamiento pueden contener más imágenes de una clase que de otra. En total, los datos de entrenamiento contienen exactamente 5,000 imágenes de cada clase.

La Figura 5.3 muestra un ejemplo aleatorio del conjunto de datos. Cada uno de los archivos por lotes contiene un diccionario con los siguientes elementos:

- **datos**: una matriz de 10,000x3,072. Cada fila de la matriz almacena una imagen en color (3 canales) de 32x32. Las primeras 1024 entradas contienen los valores del canal rojo, las siguientes 1024 son las verdes y las 1024 finales son las azules. La imagen se almacena en orden de fila principal, de modo que las primeras 32 entradas de la matriz son los valores del canal rojo de la primera fila de la imagen.

- **etiquetas**: una lista de 10,000 números en el rango 0-9. El número en el índice i indica la etiqueta de la i ésima imagen en los datos de la matriz .

Las clases son completamente excluyentes, por ejemplo no hay superposición entre automóviles y camiones: "Automóvil" incluye berlinas, SUVs, cosas de ese tipo. "Camión" incluye solo camiones grandes. Tampoco incluye camionetas.

Se obtiene un 18% de error de prueba sobre el conjunto original y un 11% cuando se usan técnicas de aumento de datos (*data augmentation*) para combatir el desbalanceo.

El conjunto de *CIFAR-100* es similar al *CIFAR-10*, con la excepción de que tiene 100 clases con 600 imágenes cada una.

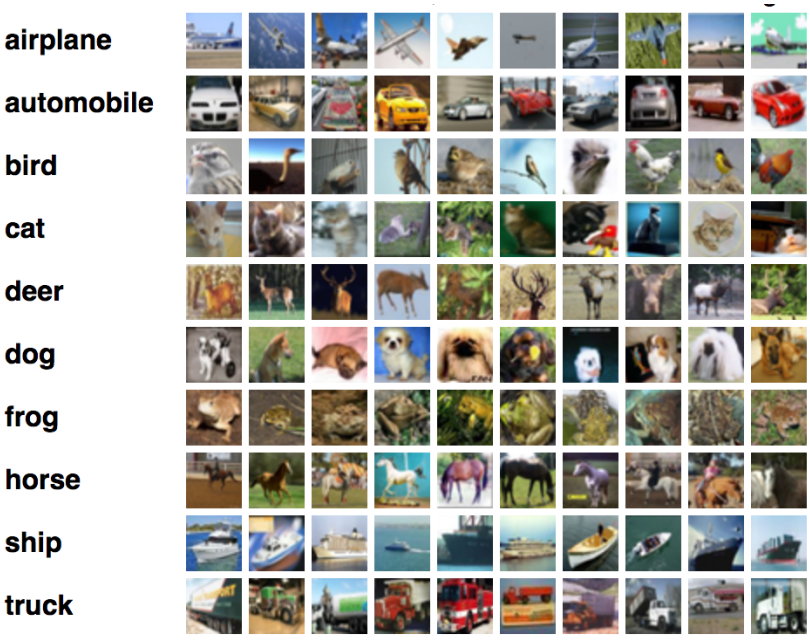


Figura 5.3: Subconjunto CIFAR10

5.2 Modelos

En este proyecto se usaron dos arquitecturas muy conocidas en la clasificación de imágenes: modelo *ResNet* y el modelo *Inception*.

5.2.1 ResNet50

ResNet es una arquitectura introducida por *Microsoft* [25] que hace posible entrenar cientos o incluso miles de capas y aún así lograr un rendimiento convincente. Esta arquitectura ha mejorado el rendimiento de muchas aplicaciones en campos como detección de objetos, reconocimiento facial y clasificación de imágenes, que es el que nos abarca.

Se parte de la siguiente premisa:

- Aumentar la profundidad de la red aumenta el problema del **Vanishing Gradient**: a medida de que el gradiente se propaga hacia atrás (hacia las capas anteriores), la multiplicación repetida puede hacer que el gradiente sea infinitamente pequeño. Como resultado, a medida que la red es más profunda, su rendimiento se satura o incluso comienza a degradarse rápidamente.

La idea de *ResNet* (Figura 5.4), se basa en aumentar el número de capas introduciendo una conexión residual (con una capa identidad). Esta capa pasa a la siguiente directamente, mejorando el proceso de aprendizaje y solucionando el gradiente de fuga.

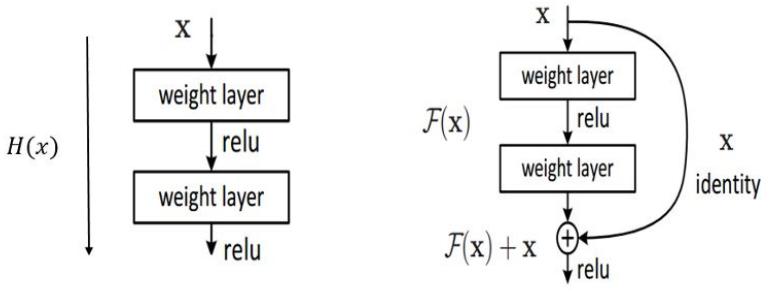


Figura 5.4: Idea principal de *ResNet*.

Estas capas residuales son explotadas por el *backpropagation*, ya que un gradiente que pueda ser "olvidado" por la red, se recupera en estas capas amplificando lo aprendido a medida que se aprende el espacio de características de la imagen de entrada.

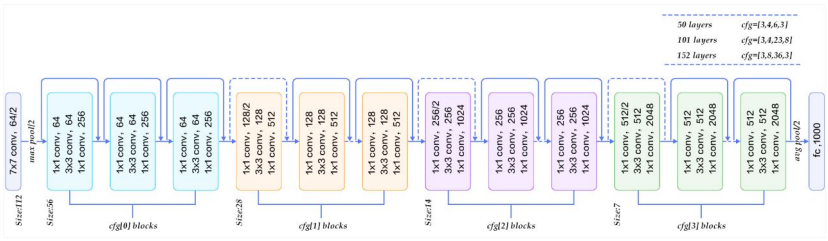


Figura 5.5: Arquitectura *ResNet*.

Él modelo *ResNet-50* es una red que tiene 50 capas de profundidad, y consta de 4 etapas, cada una con un bloque de convolución e identidad. Cada bloque de convolución tiene 3 capas de convolución. Se puede ver su arquitectura en la Figura 5.5.

5.2.2 Inception-V3

La arquitectura *Inception* fue introducida por primera vez por Szegedi en su artículo de 2014 [26]. El nombre original de esta arquitectura era *GoogLeNet* (*Inception V1*), pero las posteriores implementaciones simplemente le llamaron *Inception-VN*, donde N se refiere al número de versión publicado por *Google*.

Se tienen en cuenta las siguientes premisas:

- Las partes que no tienen interés en una imagen pueden tener tamaños diferentes (ver Figura 5.6).

Figura 5.6: De izquierda a derecha: el perro ocupa menor porcentaje de la imagen.

- La gran variación en la ubicación del objeto de interés influye en el tamaño del *kernel* a utilizar en las capas de convolución. Un *kernel* más grande implica que la información se distribuye más globalmente, y por el contrario, un *kernel* más pequeño distribuye la información más localmente.

- Las redes muy profundas son propensas al *overfitting*. Y también es difícil propagar el gradiente a través de toda la red.

- Aplicar grandes operaciones de convolución es computacionalmente costeso.

Como solución, *Inception* propone utilizar filtros de diferente tamaño en el mismo nivel. La red se volverá más ancha en lugar de más profunda. En la Figura 5.7, vemos una convolución con 3 tamaños diferentes de filtros (1x1, 3x3 y 5x5). Además, también realiza agrupación máxima. Las salidas de concatenan y se envían al siguiente módulo.

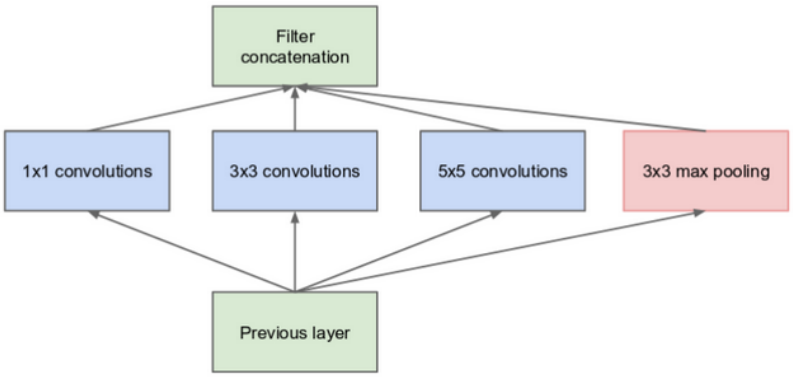


Figura 5.7: Arquitectura *GoogLeNet* (*Inception-V1*).

GoogLeNet tiene 9 módulos *Inception* apilados linealmente, 22 capas de profundidad (27, incluidas las capas de agrupación) y utiliza *average pooling* al final del ultimo módulo *Inception*.

Inception-V2 y *Inception-V3* presentan una serie de actualizaciones que aumentaron la precisión y redujeron la complejidad computacional. En concreto, *Inception-V3* demostró que alcanza una exactitud superior al 78,1% en el conjunto de datos de *ImageNet*. El modelo está formado por bloques simétricos y asimétricos que incluyen convoluciones, *average pooling*, concadenaciones, *dropout* y *full connected layers*. El optimizador de normalización por *batch* se utiliza durante todo el modelo y se aplica a las entradas de activación. La pérdida se calcula a través de *Softmax*. En la Figura 5.8 se muestra un diagrama de alto nivel del modelo.

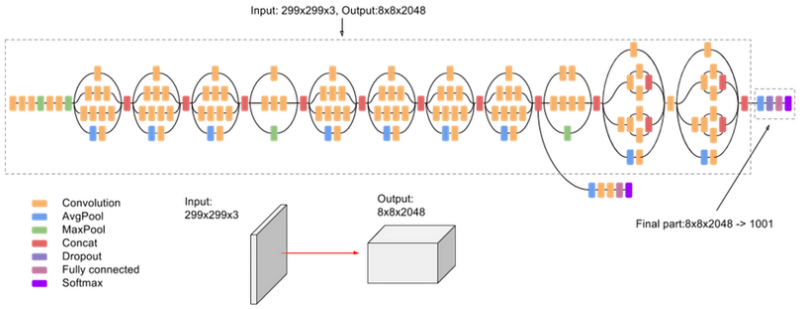


Figura 5.8: Arquitectura *Inception-V3*.

5.2.3 Entrenamiento del modelo

A continuación, entrenamos ambas arquitecturas para luego utilizar los modelos y tratar de engañarlos. El objetivo es que bajo las condiciones de robustez del estado del arte obtengamos el modelo sea un que sepa generalizar acorde a los parámetros de *loss* y *accuracy*. El resultado de nuestro entrenamiento se puede ver en la Figura 5.9). Los valores de *loss* y *accuracy* cuadran con la definición de modelo robusto mencionado: en todos los modelos, la *loss* es baja y la precisión es superior a un 80% de acierto. A continuación, vamos a probar si con nuestra nueva aproximación de medir la robustez, esto es realmente cierto.

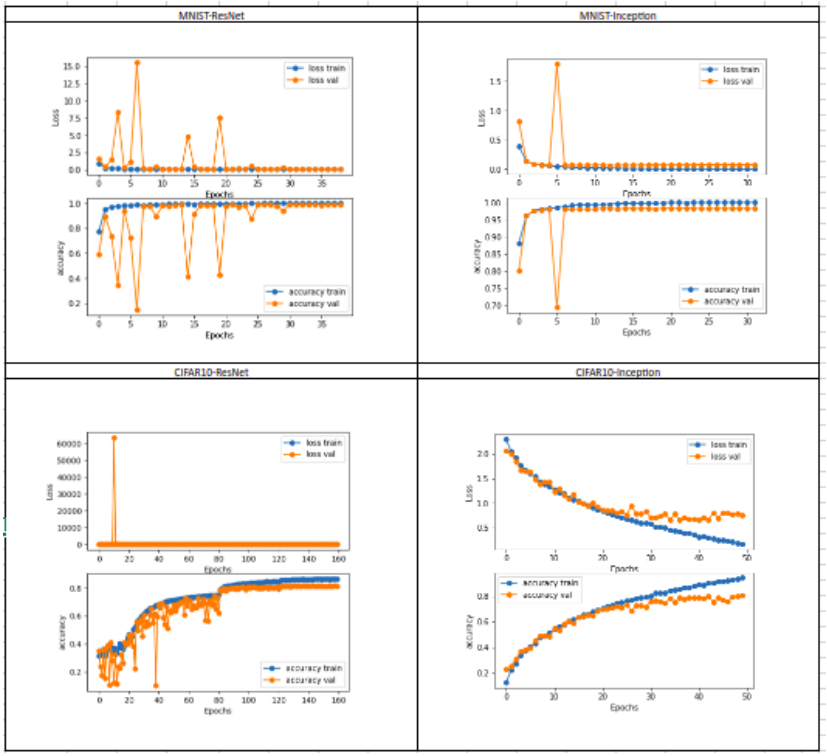


Figura 5.9: Resultados del entrenamiento del modelo ResNet e Inception con MNISTy CIFAR10

5.3 Evaluación de parámetros

Como se puede ver en el pseudocódigo de DENA (algoritmo 1) hay 3 parámetros configurables para controlar las iteraciones (parámetro $extraIters$) y las regularizaciones (parámetros λ_1 y λ_1 respectivamente) que se usan en la fase de colapso forzando al ejemplo a ser lo más similar posible a la solución real.

Pero, no sabemos que valores son los adecuados para obtener resultados óptimos a la hora de generar ejemplos adversos, por ello, antes de empezar con las aproximaciones, se es-

tudia como varían las distancias y el tiempo en función de los parámetros $extraIters$, λ_1 y λ_2 respecto a *DENA*. Se pretende hacer gráficas variando estos parámetros, con los *dataset MNIST* y *CIFAR10*, *Fashion-MNIST* y *CIFAR100* (ver sección: 5.1). Cabe destacar que es necesario entrenar de nuevo, los modelos de las Figuras 4.1 y 4.2 (ver sección 4.1) para los *datasets Fashion-MNIST* (sólo varían los datos de entrada respecto a *MNIST*, con un entrenamiento es suficiente) y *CIFAR100* (que el número de clases objetivo pasan a ser 100 en lugar de 10 de *CIFAR10*).

Para la realización de todas las gráficas se escogió la ejecución con los parámetros del Cuadro 5.1.

Dataset	Architecture	Opt	Attack	NTargeted	Thresh	lr
MNIST, CIFAR10, FASHION-MNIST, CIFAR100	ResNet	Adam	targeted	1	75%	1^{-4}

Cuadro 5.1: Configuración de las ejecuciones para el estudio de parámetros

5.3.1 Consideraciones previas

Antes de hacer un estudio de los parámetros, debemos pararnos a pensar que significa cada métrica que vamos a medir (ver sección 3.3).

En todas las ejecuciones se estudia el tiempo necesario que emplea el algoritmo para generar imágenes adversas, en el caso del parámetro $extraIters$, como cabe de esperar: la ejecución es lineal, de forma que cuantas más iteraciones de DENA, mas tiempo puede emplear en buscar el ejemplo mínimo. Y en el caso de del estudio de las λ el tiempo fluctúa ya que λ_2 varía más rápidamente que λ_1, generando tiempos de ejecución más bajos con ambos parámetros bajos.

5.3.2 Efecto del parámetro $extraIters$ sobre el resultado

Se propone ver como el parámetro $extraIters$ afecta al resultado final. $extraIters$ controla el número de iteraciones que ejecuta DENA con las regularizaciones l1 y l2, esto es, una vez que ya se ha encontrado un ejemplo adverso para los datos de la entrada. La ejecución se hace variando el parámetro $extraIters$. Se harán ejecuciones del algoritmo DENA (ver algoritmo 1) con $extraIters = 0$, 100, 250, 500 y 1000.

Las gráficas de error del parámetro $extraIters$ pueden verse en las Figuras 5.10, 5.11, 5.12 y 5.13. En estas gráficas son generadas por un *dataset MNIST*, modelo ResNet, optimizador *adam* y función de coste use_thresh_value se imprime el valor de la distancia máxima, mínima, media y mediana sobre de las distancias de todas las imágenes del *dataset*. De forma que la primera imagen de cada una de las figuras, se corresponde con la $norma_0$, la segunda imagen la $norma_1$ y la tercera imagen la $norma_2$. La $norma_\infty$ no se muestra porque en todos los casos es lineal, es decir, el porcentaje que se varió en cada píxel es muy bajo en todos

los *datasets* considerados. De hecho, el comportamiento de DENA para los diferentes *datasets* produce una curva similar, quedando probado que es independiente del conjunto de datos.

Como era de esperar, las distancias son menores en los *datasets* con una tamaño de 28x28x1, ya que tienen de media menos de 30 píxeles que modificar frente a los 126 de los *datasets* de un tamaño 32x32x3 donde el canal de color tiene un papel importante además de la diferencia de tamaño.

Se puede ver como la cantidad de píxeles modificado decrece en función del parámetro $extraIters$ ya que en todos los conjuntos de datos se modifican casi todos los píxeles de la imagen con un $extraIters = 0$, de media modifica 700 de 784 píxeles en MNIST (aproximadamente un 89%) y 3000 de 3072 en CIFAR-10 (aproximadamente un 98%). Sin embargo, con con un $extraIters = 1000$ para MNIST se modifican aproximadamente un 57% mientras que en CIFAR-10 un 48%. Como se ve, aumentando $extraIters = 1000$ las imágenes resultado son más cercanas, pero esto repercute mucho en el tiempo requerido (ver Figura 5.14) Con un $extraIters = 1000$ para MNIST la ejecución dura unos 1450 segundos (unos 24 minutos aproximadamente) y para CIFAR10 unos 6000 segundos (aproximadamente 1 hora y 40 minutos). Por defecto, decidimos $extraIters = 250$ porque equilibra bastante el resultado con el tiempo.

- **Ejecución**: Comparativa entre *datasets* (*MNIST*, *CIFAR-10*, *fashion-MNIST* y *CIFAR-100*) en un modelo *ResNet*, con *adam* y TV

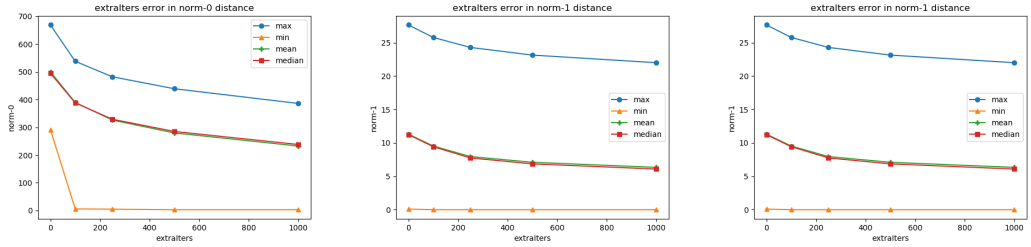


Figura 5.10: Efecto del parámetro $extraIters$ en el *dataset*, *MNIST*

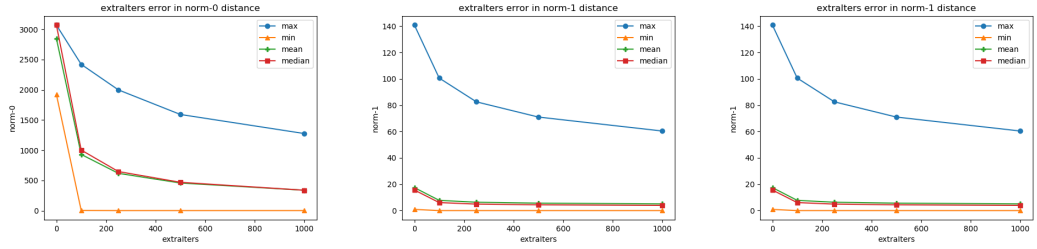


Figura 5.11: Efecto del parámetro $extraIters$ en el *dataset*, *CIFAR-10*

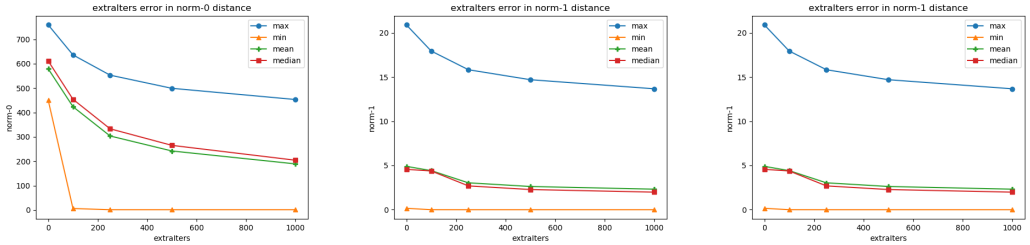


Figura 5.12: Efecto del parámetro $extraIters$ en el *dataset*, *fashion-MNIST*

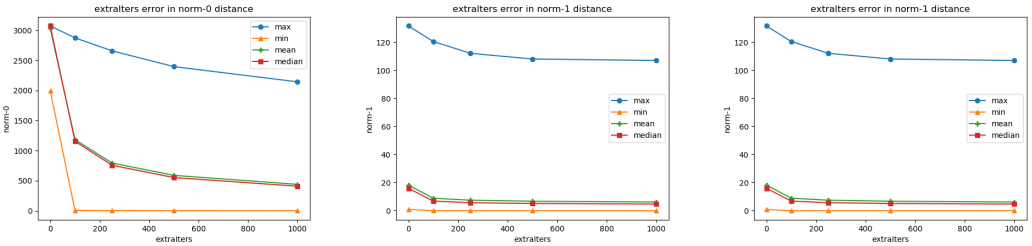
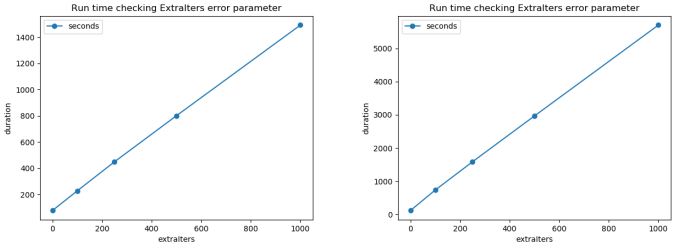


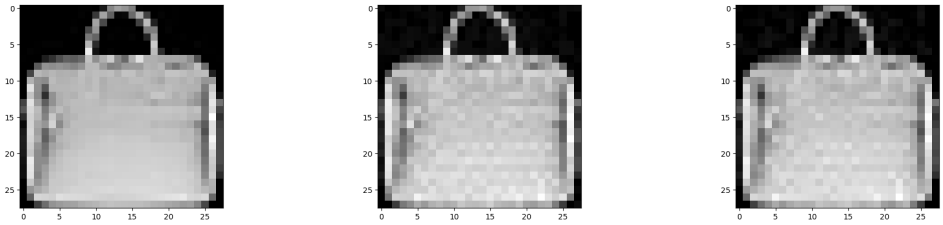
Figura 5.13: Efecto del parámetro $extraIters$ en el *dataset*, *CIFAR-100*



(a) MNIST (b) CIFAR10

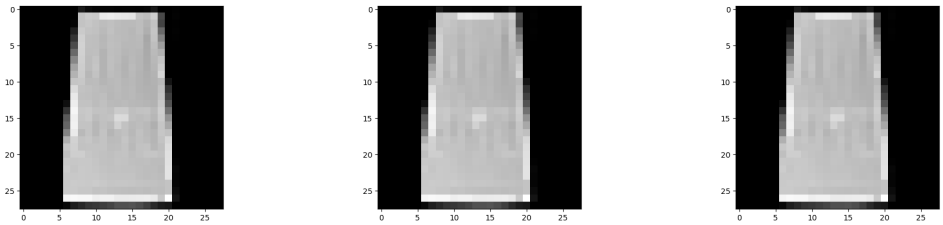
Figura 5.14: El tiempo de ejecución aumenta al aumentar el parámetro $extraIters$

En la figura 5.15 se muestran el resultado de aplicar DENA en la imagen (a) original con diferentes $extraIters$. Esta es la imagen con más distancia de las 1000 del *dataset*. Se puede apreciar que el cambio es mayor con $extraIters = 0$ que con $extraIters = 1000$. En la figura 5.16 se muestra el resultado de la imagen con menor distancia. Incluso aunque se puede apreciar diferencia en la imagen con más distancia, para nosotros, la imagen sigue siendo una maleta, y no un pantalón.



(a) Original (b) $extraIters = 0$ (c) $extraIters = 1000$

Figura 5.15: La predicción de la imagen (a) 'Bag', la (b) 'Trouser' y la (c) 'Trouser'



(a) Original (b) $extraIters = 0$ (c) $extraIters = 1000$

Figura 5.16: La predicción de la imagen (a) 'Bag', la (b) 'Shirt' y la (c) 'Shirt'

5.3.3 Efecto de los parámetros λ_1 y λ_2 sobre el resultado

La ejecución se hace variando y combinando los parámetros λ_1 y λ_2 como se muestra en el Cuadro 5.2. Las gráficas de error se pueden ver en las Figuras 5.17, 5.18, 5.19 y 5.20.

Como en el caso del estudio del parámetro $extraIters$ la distancia es mucho más alta en los *datasets* tipo CIFAR que los MNIST y en este caso va variando en función de los λ, siendo λ_1 el parámetro que tiene más peso. Las gráficas también forman una curva similar como en el caso del error en $extraIters$, por lo cual, queda demostrado que la generación de imágenes adversas en independiente del conjunto de datos usado.

Las gráficas convergen hacia valores de λ_1 y λ_2 bajos, en torno a 0.1 generando imágenes más cercanas a las originales.

λ1	0,1	0,1	0,1	0,1	0,01	0,01	0,01	0,01	0,001	0,001	0,001	0,001	0,0001	0,0001	0,0001	0,0001
λ2	0,1	0,01	0,001	0,0001	0,1	0,01	0,001	0,0001	0,1	0,01	0,001	0,0001	0,1	0,01	0,001	0,0001

Cuadro 5.2: Configuración de las ejecuciones con los parámetros λ

- **Ejecución**: Comparativa entre *datasets* (*MNIST*, *CIFAR-10*, *fashion-MNIST* y *CIFAR-100*) en un modelo *ResNet*, con *adam* y TV

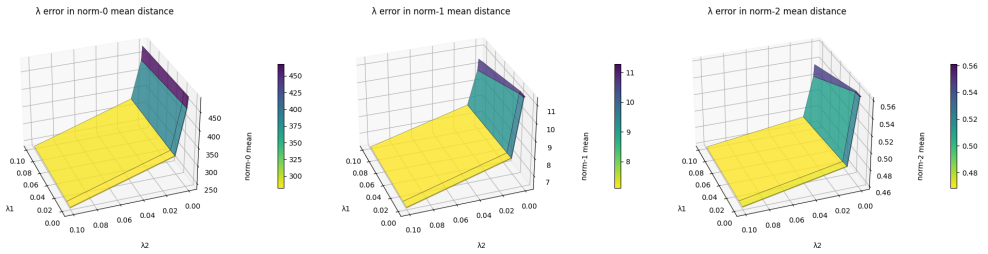


Figura 5.17: Efecto del parámetro λ en el *dataset MNIST*

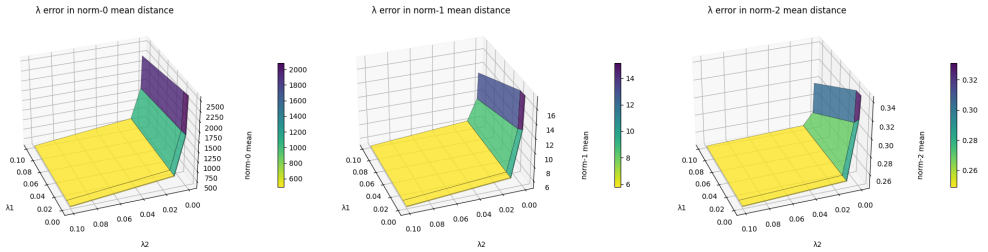


Figura 5.18: Efecto del parámetro λ en el *dataset CIFAR-10*

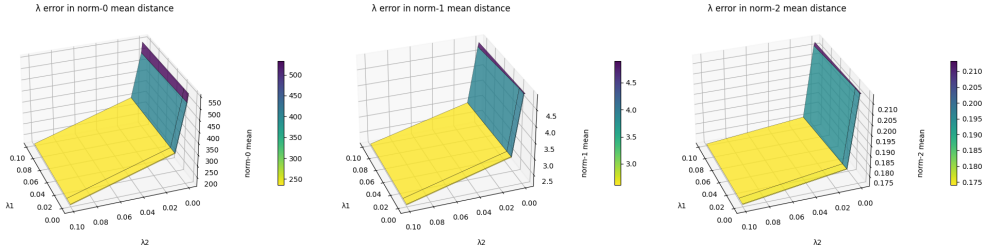


Figura 5.19: Efecto del parámetro λ en el *dataset fashion-MNIST*

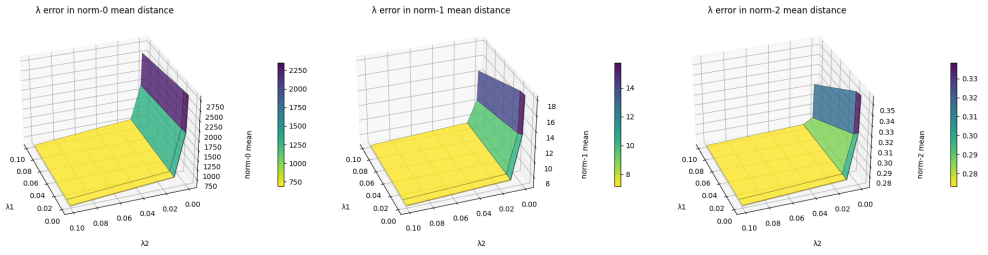


Figura 5.20: Efecto del parámetro λ en el *dataset CIFAR-100*

5.4 Primera Aproximación: *untargeted attacks*

Una vez hecho el estudio, tenemos seguridad de como afectan los parámetros variables al algoritmo, y confirmar que la estimación de la sección 4.6 sea correcta para obtener las imá-

genes más cercanas posible, pasamos a realizar la primera aproximación, que se corresponde con la ejecución de *untargeted-attacks*: busca que la salida de DENA (ver algoritmo 1) sean imágenes adversas donde la clase ganadora esté entre las N más probables por encima de la clase verdadera. Para ello, se usan las funciones de coste f2 y f3 (ver sección 3.2.1).

5.4.1 Estudio de la función de coste

Como primera comparativa, se ejecuta un modelo *ResNet* sobre el *dataset MNIST*, un optimizador *adam* y un $n = 1$ para ver como afecta la elección de la función de coste (ver Cuadro 5.7), donde los resultados son similares en ambas funciones, aunque ligeramente más cambio y cantidad de píxeles modificados en la función de coste $F3$ pero con un porcentaje de cambio casi igual. La diferencia de tiempo entre $F2$ y $F3$ es de 3 segundos más para $F3$, la cual es despreciable.

- **Ejecución**: *ResNet*, *MNIST*, *adam*, $F3$, $N = 1$

Lost f	max	min	mean	median
F2	20,1339	0,01783	7,0689	7,0157
F3	22,7339	0,1236	9,6497	9,6965

Cuadro 5.3: $norm_1$

Lost f	max	min	mean	median
F2	1,2778	0,00478	0,4169	0,4103
F3	1,3358	0,0063	0,4787	0,4747

Cuadro 5.4: $norm_2$

Lost f	max	min	mean	median
F2	484	16	335	339
F3	486	177	357	360

Cuadro 5.5: $norm_0$

Lost f	max	min	mean	median
F2	0,1227	0,0003	0,0275	0,0264
F3	0,1250	0,0003	0,0276	0,0264

Cuadro 5.6: $norm_\infty$

Cuadro 5.7: La función de coste $F3$ produce distancias ligeramente mayores que la función de coste $F2$.

5.4.2 Estudio del parámetro N

Si variamos la cantidad de exigencia en a la hora de escoger la clase *targeted* (N más alto), las distancias son más altas (ver Cuadro 5.12), porque el modelo tienen más claro que esa clase no es la clase correcta, así que se necesita modificar mucho más los datos de entrada para engañarlo. Esto también se ve en los tiempos de ejecución (ver Cuadro 5.13), los tiempos son mayores para N y $F3$.

- **Ejecución**: *ResNet*, *MNIST*, *adam*, $F3$

N	max	min	mean	median
1	22,7339	0,1236	9,6497	9,6965
3	66,3043	0,1224	14,5602	14,3059

Cuadro 5.8: $norm_1$

N	max	min	mean	median
1	1,3358	0,0063	0,4787	0,4747
3	7,0115	0,0062	0,8307	0,7946

Cuadro 5.9: $norm_2$

N	max	min	mean	median
1	486	177	357	360
3	735	35	414	425

Cuadro 5.10: $norm_0$

N	max	min	mean	median
1	0,1250	0,0003	0,0275	0,0264
3	1,0000	0,0003	0,1002	0,0801

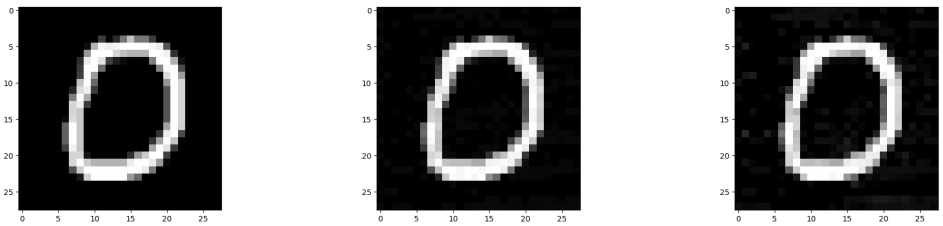
Cuadro 5.11: $norm_\infty$

Cuadro 5.12: El aumento del parámetro N implica distancias más grandes.

N	Runtime
1	0:07:25
3	0:22:18

Cuadro 5.13: El tiempo de ejecución aumenta a medida que aumenta N.

En la Figura 5.21 se muestra un ejemplo visual del *dataset MNIST* en que el cambio del parámetro N es apenas imperceptible entre ambas imágenes.



(a) Original, clase:'0' (b) $N = 1$, clase:'2' (c) $N = 3$. clase:'4'

Figura 5.21: Ejemplo de ejemplos adversos cambiando variando N

5.4.3 Estudio del optimizador

La función de coste $F3$ da resultados más cercanos en un tiempo ligeramente menor, vamos a ver como se comporta en distintos optimizadores (ver Cuadro 5.17). La media de cambio es mayor con un optimizador tipo *adam* y similares entre *momentum* y *sgd*, siendo mayor en *momentum*. En cuanto a los píxeles modificados esta comparación se mantiene para *adam*, pero *sgd* pasa a modificar más píxeles de media que *momentum*, siendo de nuevo valores similares. En el cuadro 5.17 podemos ver que aunque los cuadros 5.14,5.15 y 5.16 marquen

opiniones negativas contra el optimizador *adam*, el ratio de cambio en cada píxel es mucho menos que en *sgd* y *momentum*. Además, el tiempo de ejecución (ver Cuadro 5.19) en *adam* menor.

- **Ejecución**: *ResNet*, *MNIST*, $F3$, $N = 1$

opt	max	min	mean	median
sgd	17,8124	0,0603	6,2251	6,1045
mometum	17,4963	0,0539	6,2533	6,1397
adam	22,7339	0,1236	9,6497	9,6965

Cuadro 5.14: $norm_1$

opt	max	min	mean	median
sgd	1,7272	0,0040	0,4343	0,4203
mometum	1,5309	0,0036	0,4173	0,4048
adam	1,3358	0,0063	0,4787	0,4747

Cuadro 5.15: $norm_2$

opt	max	min	mean	median
sgd	548	27	282	298
mometum	557	34	272,98	289
adam	486	177	357,126	360

Cuadro 5.16: $norm_0$

opt	max	min	mean	median
sgd	0,6100	0,0008	0,0840	0,0781
mometum	0,3585	0,0007	0,0797	0,0765
adam	0,1250	0,0003	0,0275	0,0264

Cuadro 5.17: $norm_\infty$

Cuadro 5.18: El optimizador *adam* genera distancias más altas frente al resto, pero en porcentaje de modificación en los píxeles es menor.

Lost f	Runtime
sgd	0:22:18
mometum	0:23:09
adam	0:07:25

Cuadro 5.19: El optimizador adam es el más rápido de los tres.

5.4.4 Comparativa entre arquitecturas y *datasets*

En este caso, por motivos de rapidez, se elige el optimizador *adam* ya que aunque da resultados peores, realmente estas métricas son consistentes. El objetivo es comparar diferentes métricas, no que los resultados sean los mínimos posibles y de esta forma estudiar la robustez. En el Cuadro 5.24, se puede ver que una arquitectura *Inception* de media modifica más las imágenes frente a *ResNet* pero el número de cambio de píxeles y el porcentaje de cambio es mayor en *ResNet*. En cuanto a los *datasets*, la media de cambio en *MNIST* y el porcentaje de cambio es mayor pero el número de píxeles a modificar es menor que un *dataset CIFAR10*.

- **Ejecución**: Comparativa de arquitectura y *dataset* con: $F3$, $N = 1$, $opt = adam$

Dataset	Network	max	min	mean	median
MNIST	ResNet	22,7339	0,1236	9,6497	9,6965
	Inception	84,5158	0,1452	30,5916	29,8616
CIFAR10	ResNet	67,3058	0,7767	7,5070	7,0330
	Inception	64,7688	0,8127	11,9048	11,8453

Cuadro 5.20: $norm_1$

Dataset	Network	max	min	mean	median
MNIST	ResNet	1,3358	0,0063	0,4787	0,4747
	Inception	4,2952	0,0068	1,6212	1,5939
CIFAR10	ResNet	2,1789	0,0157	0,2523	0,2191
	Inception	2,1517	0,0160	0,3951	0,3303

Cuadro 5.21: $norm_2$

Dataset	Network	max	min	mean	median
MNIST	ResNet	486	177	357	360
	Inception	514	152	337	342
CIFAR10	ResNet	1458	264	574	654
	Inception	1387	218	484	553

Cuadro 5.22: $norm_0$

Dataset	Network	max	min	mean	median
MNIST	ResNet	0,1250	0,0003	0,0275	0,0264
	Inception	0,2489	0,0003	0,0956	0,0949
CIFAR10	ResNet	0,0855	0,0003	0,0061	0,0049
	Inception	0,0587	0,0003	0,0082	0,0063

Cuadro 5.23: $norm_\infty$

Cuadro 5.24: Un modelo *Inception* produce resultados altos en las distancias a costa de menor cambio en los píxeles.

En cuanto al tiempo de ejecución (ver Cuadro 5.25) una vez más se enfatiza la dificultad de obtener ejemplos adversos para un *dataset* tipo *CIFAR10* y una arquitectura *Inception* en comparación con un modelo *ResNet MNIST*. Por lo cual podemos deducir que para un *untargeted attack* los modelos *Inception CIFAR10* son más robustos que los modelos *ResNet MNIST*. Además de la clara diferencia de tiempos entre los optimizadores, siendo *adam* el más rápido.

MNIST	Resnet	SGD	0:22:18
		Momentum	0:23:09
		Adam	0:07:25
	Inception	SGD	0:28:08
		Momentum	0:29:18
		Adam	0:08:14
CIFAR10	Resnet	SGD	0:49:49
		Momentum	0:51:49
		Adam	0:23:40
	Inception	SGD	1:16:52
		Momentum	1:18:58
		Adam	0:35:41

Cuadro 5.25: El tiempo de ejecución de *adam*, supera al de los demás optimizadores sin importar el modelo o dataset.

5.5 Segunda Aproximación: *targeted attacks*

En la segunda aproximación se llevan a cabo *targeted attacks*: el objetivo es que gane una clase en concreto. En este caso, la clase N más probable, de las predicciones de cada imagen antes de ser modificada. Además, una imagen adversa sólo será considerada válida cuando la seguridad de esa clase esté por encima de un valor denominado $thresh$. Para ello, se usan las funciones de coste *CE* y *TV* (ver sección 3.2.1).

5.5.1 Estudio de la función de coste

Como en la aproximación anterior, vamos a estudiar en un primer momento como afecta la función de coste a las distancias (ver Cuadro 5.30).

- **Ejecución**: *ResNet*, *MNIST*, $opt = adam$, $thresh = 0.75$

Lost f	max	min	mean	median
CE	27,6761	0,1034	11,0173	10,9363
TV	24,2802	0,0059	7,9559	7,7522

Cuadro 5.26: $norm_1$

Lost f	max	min	mean	median
CE	1,5971	0,0057	0,5478	0,5362
TV	1,5088	0,0025	0,4819	0,4653

Cuadro 5.27: $norm_2$

Lost f	max	min	mean	median
CE	627	167	449	447
TV	483	5	326	329

Cuadro 5.28: $norm_0$

Lost f	max	min	mean	median
CE	0,1781	0,0003	0,0378	0,0348
TV	0,1457	0,0003	0,0332	0,0308

Cuadro 5.29: $norm_\infty$

Cuadro 5.30: La función use_thresh_value obtiene, de manera clara, ejemplos adversos mucho más parecidos a los datos de entrada.

La media de cambio de las imágenes es superior en la función de coste $categorical\_cross\_entropy$ que en use_thresh_value, así como la cantidad de píxeles modificados, volviendo a ser despreciable la diferencia entre el porcentaje que se varía.

En cuanto al tiempo de ejecución en este caso será de 30 segundos más para CE, por lo cual, es independiente el tiempo a la hora de escoger una métrica. Se elige use_thresh_value ya que las distancias que genera son menores para continuar con el estudio.

5.5.2 Estudio del parámetro $thresh$

En la primera aproximación (ver sección 5.4) se hicieron pruebas con $n_{targeted} = 1$ y $n_{targeted} = 3$. En esta segunda aproximación, $n_{targeted}$ también se usa pero cambia su significado (ver algoritmo 4.2.3). Todas las ejecuciones de tipo *targeted attacks* se hicieron con $n_{targeted} = 1$, es decir, $yTarget\_N$ es un vector en *one-hot-encoding* con el valor de 1 en la clase más probable sin contar con la clase verdadera antes de empezar a modificar la imagen. En este caso el para ser más exigentes con la clase objetivo, es necesario declarar un nuevo parámetro: $thresh$ (ver sección 4.2.2). Además de que la clase ganadora esté un $thresh\%$ segura de que es la clase verdadera, esté parámtro se usa en la función de coste use_thres_value (ver sección 3.2.1).

- **Ejecución**: *ResNet*, *MNIST*, $opt = adam$, $loss\_f = TV$

thresh	max	min	mean	median
0.75	24,2802	0,00594821	7,95589	7,75223
0.95	30,6828	0,0331356	10,6502	10,2782

Cuadro 5.31: $norm_1$

thresh	max	min	mean	median
0.75	1,5088	0,0025	0,4819	0,4653
0.95	2,0449	0,0083	0,6160	0,5862

Cuadro 5.32: $norm_2$

thresh	max	min	mean	median
0.75	483	5	326,048	329
0.95	506	8	365,994	367

Cuadro 5.33: $norm_0$

thresh	max	min	mean	median
0.75	0,1457	0,0003	0,0332	0,0308
0.95	0,2895	0,0007	0,0458	0,0403

Cuadro 5.34: $norm_\infty$

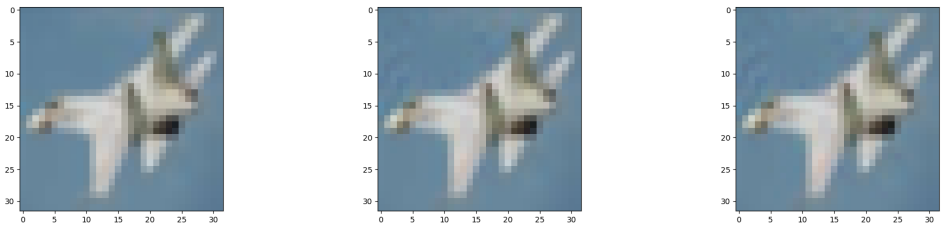
Cuadro 5.35: Aumentar el el valor de confianza en la clase *target* aumenta la diferencia entre las imágenes.

El Cuadro 5.35 nos confirma como con más exigencia las distancias son mayores, aunque el tiempo es casi despreciable como puede verse en el Cuadro 5.36. Un $thresh = 0.95$ hace más cambios en las imágenes pero con un cambio similar en cuanto a número de píxeles modificados y cantidad de modificación.

thresh	Runtime
0,75	0:07:35
0,95	0:08:18

Cuadro 5.36: La diferencia de tiempo al aumentar la exigencia de confianza en la clase *target* no es significativa.

En la Figura 5.22 se muestra un ejemplo visual del *dataset CIFAR10* en que el cambio del parámetro $thresh$ es apenas imperceptible entre ambas imágenes.



(a) Original, clase: 'airplane' (b) $thresh = 0.75$, clase: 'bird' (c) $thresh = 0.95$, clase: 'bird'

Figura 5.22: Ejemplo de ejemplos adversos cambiando variando $thresh$

5.5.3 Estudio del optimizador

En la aproximación anterior el optimizador más rápido era *adam* mientras que *momentum* y *sgd* tenían unos tiempos similares, pero como actuaban notablemente diferente sobre

las imágenes, vamos a ver como se comportan en caso de los *targeted attacks* y si su comportamiento es el mismo.

- **Ejecución**: *ResNet*, *MNIST*, $thresh = 0.75$, $loss\_f = TV$

opt	max	min	mean	median
sgd	22,5380	0,0065	5,1793	5,0515
momentum	17,1732	0,0064	5,2067	5,0633
adam	24,2802	0,0059	7,9559	7,7522

Cuadro 5.37: $norm_1$

opt	max	min	mean	median
sgd	3,7283	0,0014	0,4865	0,4540
momentum	2,8650	0,0019	0,5036	0,4784
adam	1,5088	0,0025	0,4819	0,4653

Cuadro 5.38: $norm_2$

opt	max	min	mean	median
sgd	532	11	218,739	203
momentum	525	11	208,043	188
adam	483	5	326,048	329

Cuadro 5.39: $norm_0$

opt	max	min	mean	median
sgd	0,8133	0,0005	0,1012	0,0890
momentum	0,8039	0,0004	0,0961	0,0902
adam	0,1457	0,0003	0,0332	0,0308

Cuadro 5.40: $norm_\infty$

Cuadro 5.41: Un optimizador sgd y *momentum* generan imágenes similares, frente a imágenes más distantes con un optimizador adam.

Si comparamos los resultados del Cuadro 5.42 vemos que de nuevo el optimizador que hace más cambios es *adam*, frente a *momentum* y *sgd* con resultados similares y ligeramente mayores en *momentum*, en cambio la cantidad de píxeles modificados y la cantidad de cambio es mayor en *sgd*, aun siendo similares. El comportamiento para un *targeted attack* es exactamente igual a *untargeted attacks* en cuanto al optimizador y en cuanto a tiempos (ver Cuadro 5.42).

opt	Duration (s)
sgd	0:24:32
momentum	0:25:42
adam	0:07:35

Cuadro 5.42: Escoger un optimizador *adam* será clave para ejecuciones rápidas.

5.5.4 Comparativa entre arquitecturas y *datasets*

Finalmente, para terminar con el análisis, se compararán las arquitecturas y *datasets* con los datos recogidos de los apartados anteriores: un $thresh = 0.75$, $loss_f = TV$ y un $opt = adam$ por rapidez e imágenes más similares.

- **Ejecución**: Comparativa de arquitectura y *dataset* con: TV, $opt = adam$, $thresh = 0.75$

Dataset	Network	max	min	mean	median
MNIST	ResNet	24,280	0,006	7,956	7,752
	Inception	87,287	0,014	29,777	28,643
CIFAR10	ResNet	82,554	0,001	6,339	4,923
	Inception	80,276	0,001	12,224	9,893

Cuadro 5.43: $norm_1$

Dataset	Network	max	min	mean	median
MNIST	ResNet	1,509	0,003	0,482	0,465
	Inception	4,313	0,006	1,683	1,644
CIFAR10	ResNet	2,287	0,000	0,252	0,217
	Inception	2,189	0,001	0,462	0,421

Cuadro 5.44: $norm_2$

Dataset	Network	max	min	mean	median
MNIST	ResNet	483	5	326,048	329
	Inception	500	4	347,965	353
CIFAR10	ResNet	1995	1	617,466	647
	Inception	2104	1	719,458	721

Cuadro 5.45: $norm_0$

Dataset	Network	max	min	mean	median
MNIST	ResNet	0,1457	0,0003	0,0332	0,0308
	Inception	0,2534	0,0007	0,1116	0,1104
CIFAR10	ResNet	0,1099	0,0003	0,0084	0,0069
	Inception	0,0781	0,0003	0,0138	0,0120

Cuadro 5.46: $norm_\infty$

Cuadro 5.47: Una arquitectura Inception es más robusta que una arquitectura ResNet.

En el Cuadro 5.47 se puede ver un comportamiento similar a la aproximación anterior: la arquitectura *Inception* modifica más las imágenes que *ResNet* dándonos una idea de que *Inception* es una red más robusta. En cuanto a los *datasets MNIST* necesitan un porcentaje de cambio más alto que *CIFAR10* pero la cantidad de píxeles a modificar es menor. El porcentaje de cambio es más alto en *MNIST* también.

En cuanto a los tiempos, como ya veiamos en *untargeted attacks*, empiezan a ser consi-

derablemente altos para un modelo *Inception*, *CIFAR-10* con optimizadores *sgd* o *adam* (ver Cuadro 5.48).

MNIST	Resnet	SGD	0:24:32
		Momentum	0:25:42
		Adam	0:07:35
	Inception	SGD	0:28:03
		Momentum	0:32:03
		Adam	0:08:32
CIFAR10	Resnet	SGD	0:59:37
		Momentum	1:01:31
		Adam	0:26:25
	Inception	SGD	1:46:20
		Momentum	1:48:44
		Adam	0:44:44

Cuadro 5.48: Un modelo $ResNet$ puede ser engañado en un menor tiempo, sobretodo si se usa optimizador $adam$.

Capítulo 6

Conclusiones

Como comentábamos en el primer capítulo, los ataques adversos son un problema actual y difícil de detectar. Además durante los últimos años ha habido un gran incremento de uso de *deep learning* para resolver problemas en el día a día.

El objetivo principal de este trabajo era dar un valor numérico de robustez de forma más fiable que en el estado del arte a las CNN evaluando la capacidad para soportar los ataques adversos, para ello se usaron imágenes de datasets conocidos y modelos ampliamente testeados y utilizados en aplicaciones reales actualmente.

Para medir el cambio en las imágenes fue necesario implementar un algoritmo que generara los ataques en un tiempo asequible para ser testeado. Por el cual, el cuello de botella en este proyecto estuvo continuamente ligado a optimizaciones y mejoras para acelerar el tiempo de ejecución.

Además se amplió la definición de que es "engañar" a un modelo, ya que la definición clásica de los ataques tiene puede caer en sesgos hacia una clase en concreto, en el caso de los *untargeted attack* o modificar una imagen sin importar cuanto es el cambio requerido con tal de cambiar la clase de salida, en el caso de los *targeted attack*. Por ello, en nuestro proyecto, se introducen condiciones más restrictivas para considerar que un ejemplo es válido.

Para la primera aproximación se implementó un *untargeted attack*, el cual modifica la imagen de forma que la clase ganadora en la predicción del modelo sea cualquier clase menos la clase real, y se añade la restricción de que tiene que haber N clases por encima de la clase real que tengan más confianza para que un ejemplo adverso sea considerado válido.

En la segunda aproximación se implementó un *targeted attack* que cambia la predicción a una clase elegida *a priori*, con la restricción de que solo se considerará ejemplo adversario válido si la confianza en la clase ganadora es superior a un umbral de confianza predefinido.

Además, en los ataques generados por el algoritmo DENA, tienen parámetros de regularización controlados, de forma que una vez el ejemplo es expandido hacia una región en la que se encuentran ejemplos adversos, se busca el mínimo ejemplo, para que la imágenes sea

lo más similar posible a la real.

Para la ejecución de los ataques, previamente fue necesario hacer un estudio sobre los parámetros de control del algoritmo DENA demostrando que el tiempo de ejecución está fuertemente ligado al parámetro $extraIters$ y a los parámetros λ_1 y λ_2.

El parámetro $extraIters$ le permite al gradiente tener más tiempo para intentar converger en la clase a la que se le está empujando y con ello estar cada vez más próximo a una clase diferente a la original. Y los parámetros λ_1 y λ_2 regulan la velocidad a la que se acercan a un ejemplo más similar al original, de forma que, una vez encontrado la región de ejemplos adversos, van más rápidos o menos hacia un ejemplo mínimo.

Finalmente se optó por un $extraIters = 250$ y valores de λ_1 y λ_2 con un valor de 0.1, que aún no siendo los valores para los cuales se obtenían menores distancias, si se generaban distancias bajas en tiempos de ejecución rápidos.

Después de realizar cada tipo de ataque se da un valor de la distancia entre las imágenes reales y las modificadas, quedando constancia de manera cuantitativa la similitud y el cambio necesario que hay que realizar para que una red neuronal convolucional pueda ser atacada. En este estudio, como se puede ver en los resultados, ambos ataques se comportan de manera prácticamente igual ante funciones de coste diferentes (donde el tiempo de ejecución y las distancias son similares), siendo restrictivos en la clase *target*, el comportamiento del optimizador: con *adam* los resultados se obtienen antes con un coste de distancias más altas. Finalmente, también ambas aproximaciones se comportan de manera similar para diferentes arquitecturas y *datasets*, siendo *Inception* y *CIFAR10* los modelos que más tiempo lleva encontrar un adversario y cuyas distancias son más altas, por lo cual, entre un modelo *ResNet* y un modelo *Inception*, este es el más robusto pero cuya modificación de cada píxel individual es menor.

Como trabajo futuro serían interesantes dos líneas de trabajo, por un lado, hacer robustos a los modelos ante estos ataques con algún tipo de *denoiser* a la entrada de la red como se propone en el congreso NIPS, y por otro lado, probar nuestras métricas ante modelos entrenados con defensas específicas para ataques adversos y así estudiar su robustez.

Nos gustaría que nuestros modelos puedan exhibir una confianza baja cuando operan en regiones que no han visto antes. Queremos que "fracasen" cuando se presente a la red un ejemplo adverso.

Por lo menos, el tema de los ejemplos adversos nos da una idea para tratar de defendernos de ellos, ya que antes de su aparición no teníamos ni idea de que la forma de entrenar las redes neuronales convoluciones tiene errores de concepto. Por lo tanto, lo que se proporcionó en este proyecto, es una herramienta más para evaluar los modelos y evitar posibles ataques en un mundo en el que confiamos en ellos con vidas humanas a través de automóviles autónomos y otras automatizaciones.

Lista de acrónimos

CNN *Convolutional Neural Networks.*

API *Application Programming Interface.*

GPU *Graphics Processing Unit.*

NYU *New York University.*

ReLu *Rectified Linear Units.*

SGD *Stochastic Gradient Descent.*

Adam *Adaptive Moment Estimation.*

SUV *Sport Utility Vehicle*

NIPS *Neural Information Processing Systems Foundation*

FGM *Fast Gradient Method*

LGM *Logic Gradient Method*

CGD *Class Gradient Method*

PGD *Pixel Gradient Method*

FGD *Function Gradient Method*

MI-FGSM *Iterative Momentum - Fast Gradient Sing Method*

Glosario

Deep Learning es parte de un conjunto más amplio de métodos de aprendizaje automático basados en asimilar representaciones de datos.

Python es un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional; e interpretado cuya filosofía hace hincapié en la legibilidad de su código.

Array es una estructura de programación para guardar de datos, que en *Python* tiene que ser del mismo tipo.

Open Source es el software cuyo código fuente y otros derechos que normalmente son exclusivos para quienes poseen los derechos de autor, son publicados bajo una licencia de código abierto o forman parte del dominio público.

ConvNets o red neuronal convolucional es un tipo de red neuronal artificial donde las neuronas corresponden a campos receptivos de una manera muy similar a las neuronas en la corteza visual primaria de un cerebro biológico.

Target o etiqueta, suele usarse con referencia a un objetivo, una meta o un blanco.

Loss es un número que indica qué tan mala fue la predicción del modelo en un solo ejemplo.

Batch o lote que abarca a un grupo de ejemplos de entrenamiento.

Kernel o filtro es una matriz de coeficientes por los que deben operarse los valores de intensidad del entorno de cada punto de una imagen para obtener la imagen original filtrada.

Maching o correspondencia es un proceso de asignación entre características similares.

Clipping o recorte consiste en seleccionar y sacar del origen como entidad única para su uso.

Frame cada una de las imágenes instantáneas en las que se divide un vídeo.

Graffiti es una modalidad de pintura libre, destacada por su ilegalidad, generalmente realizada en espacios urbanos.

Denoiser eliminador de ruido.

Logits es el vector de predicciones sin procesar (no normalizadas) que genera un modelo de clasificación.

Black Box Attack o ataque de caja negra es aquel donde el atacante no tiene conocimiento del modelo.

White Box Attack o ataque de caja blanca es aquel donde el atacante tiene conocimiento de alguno o todos los parámetros del modelo.

NaN : Not a Number es un tipo de dato numérico que se utiliza para representar cualquier valor que no esté definido o no se pueda representar.

Bucle for : es un bucle que repite el bloque de instrucciones un número predeterminado de veces.

Bibliografía

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [En línea]. Disponible en: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[2] "Tencent keen security lab: Experimental security research of tesla autopilot," https://keenlab.tencent.com/en/2019/03/29/Tencent-Keen-Security-Lab-Experimental-Security-Research-of-Tesla-Autopilot/, accessed: 2021-02-07.

[3] Y. Deng, X. Zheng, T. Zhang, C. Chen, G. Lou, and M. Kim, "An analysis of adversarial attacks and defenses on autonomous driving models," in *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2020, pp. 1–10.

[4] "Comma AI page," https://comma.ai/, accessed: 2021-02-07.

[5] "Comma AI github," https://github.com/commaai/openpilot, accessed: 2021-02-07.

[6] "WAYMO OPEN dataset," https://waymo.com/open/data/, accessed: 2021-02-07.

[7] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2015.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine

learning on heterogeneous systems," 2015, software available from tensorflow.org. [En línea]. Disponible en: https://www.tensorflow.org/

[9] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[10] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[11] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," 2017.

[12] "NIPS official page," https://nips.cc/, accessed: 2021-01-01.

[13] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, "Boosting adversarial attacks with momentum," 2018.

[14] F. Liao, M. Liang, Y. Dong, T. Pang, X. Hu, and J. Zhu, "Defense against adversarial attacks using high-level representation guided denoiser," 2018.

[15] "NIPS17 Adversarial learning final results," https://www.kaggle.com/google-brain/nips17-adversarial-learning-final-results, accessed: 2021-01-01.

[16] "NIPS18 Adversarial learning final results," https://www.crowdai.org/challenges/nips-2018-adversarial-vision-challenge-robust-model-track, accessed: 2021-01-01.

[17] P.-Y. Chen, Y. Sharma, H. Zhang, J. Yi, and C.-J. Hsieh, "Ead: Elastic-net attacks to deep neural networks via adversarial examples," *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[18] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM Journal on Imaging Sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[19] P.-Y. Chen, Y. Sharma, H. Zhang, J. Yi, and C.-J. Hsieh, "Ead: Elastic-net attacks to deep neural networks via adversarial examples," 2018.

[20] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, 2010.

[21] D. Cireşan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," *IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[22] H. X. y Kashif Rasul y Roland Vollgraf. Fashion-mnist: un nuevo conjunto de datos de imágenes para la evaluación comparativa de algoritmos de aprendizaje automático.

[23] "Zalando," https://jobs.zalando.com/en/tech/?gh_src=nevh2y1, accessed: 2021-14-07.

[24] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [En línea]. Disponible en: http://www.cs.toronto.edu/~kriz/cifar.html

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [En línea]. Disponible en: http://arxiv.org/abs/1512.03385

[26] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. [En línea]. Disponible en: http://arxiv.org/abs/1512.00567